university of groningen / faculty of science and engineering

Wave classification on generated data

Neural Networks Project
Academic Year 2021-2022

Thijs Lukkien (s3978389)

Group 26

Abstract

Classification tasks have a wide range of applications, from computer vision to disease recognition. The aim of this project is to create a neural network that is able to complete a simple classification task. More specifically, an echo state network is trained to classify three different wave patterns. Using a pseudorandom sequence of alternating sine, sawtooth, and square waves as input, an echo state network was trained to classify the type of wave at each timestep. Hyperparameter tuning was used in order to optimize the network performance. The results show that the trained model is able to accurately classify each wave. The final model achieved an accuracy of 86.5% with a standard deviation of 1.5% over 30 epochs.

Contents

A note on the subject

Initially, the subject that this project aimed at was classifying pathological heartbeats. Due to circumstances, the group was split up. As the workload of doing both the data (analysis, selection, cleaning, modulation) and the algorithm was too much for me, the focus lay on the algorithm. Hence, simulated data is used to train and test the algorithm. This works as required for the purpose of the algorithm, but it does limit how much one can be introduced into the subject of a project and how much one can relate back to the subject later in the paper. The report will therefore be on the short side, though one could also call it 'concise'.

I. Introduction

Classification tasks have a wide range of applications; from computer vision to disease recognition. Such classifications could be done by non-AI algorithms, however, AI algorithms have greater flexibility to adapt to changes in the input. In this project, an echo state network (ESN) is used to classify three types of wave functions. This is similar to (Jaeger, 2007), where an example was given of an ESN classifying two types of wave functions. In this project, the wave functions are generated in a pseudo-random sequence in order to easily generate different input samples.

As mentioned earlier, one could produce a non-AI algorithm that solves the same task. For example, by analyzing the slope between points. However, if the task were to change; if the wave functions for which the algorithm was written changed, then this algorithm would no longer work. Neural networks (NN) learn from input and are therefore better able to adapt to such changes.

The model output predictions will be compared to the input labels in order to determine its accuracy. Furthermore, the predictions will be plotted along with the labels in order to visually examine the results.

II. Data

The aim of this section is to describe the data that was used for the training and testing of the model. Here, it will be discussed how the input data is generated and manipulated, as well as its specifications.

Source

The data that was used in this project was psuedorandomly generated using three types of wave functions. The first is a sine wave, the second is a sawtooth function and the third is a square wave function. The first is generated using the numPy library (Harris et al., 2020) and the latter two are generated by using the signal package from the sciPy library (Virtanen et al., 2020). Each wave has a phase of 0.5π and the duration of each wave is 1000 timesteps. An example of the input signal can be seen in figure 1.

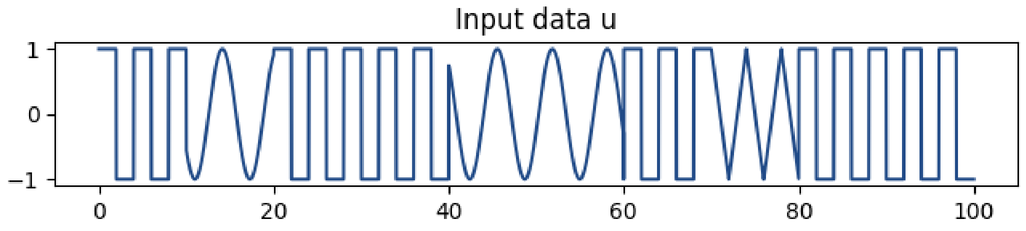

Fig. 1: Input data u^{in}. The square wave function can be observed at $T = 0$, the sine function at $T = 10$, and the sawtooth function at $T = 70$

Input specification

The timesteps are of size $t = 0.01$ and the total duration of the input is $T = 100$. This means that the input has 10000 timesteps. For every timestep t, there is an input sample $s_t = (u_t^{in}, y_t^{in})$ where u_t^{in} is the input signal and y_t^{in} is the input label. The input signal is what makes the wave, a label holds the class of wave at that timestep. These labels are one-hot encoded as the difference between low and high numerical labels does not represent their numerical difference. That is: if there were 30 classes, then a misclassification between class 1 and 2 would be equally wrong as a misclassification between class 1 and 30. This is not represented in their numerical difference as the prior misclassification is 'merely' 1 class separated from the correct answer whereas the latter is separated by 29. In one-hot encoding, the difference between class prediction and class label can be computed accordingly. An example of the input labels y^{in} can be seen in Table I.

Class 1	Class 2	Class 3
1	0	0
0	1	0
0	0	1

TABLE I: One-hot encoding for three classes.

III. Methods and experiments

The following section aims to summarize the methods used to produce the learning algorithm. All methods in this project that use pseudorandom number generation, do so via the Random package of the Numpy library (Harris et al., 2020), using seed 42.

A. The Model

For this task, an echo state network was used. This is a type of recurrent neural network (RNN) where only the output weights are trained. The weights for the input and hidden layers are pseudo-randomly initialized, after which the hidden layer states are computed.

This branch of RNNs is called 'reservoir computing'. Similar to the idea of a 'reservoir', the hidden layer states are computed without influence from the learning algorithm; they are left alone and observed. These observations (reservoir readings) are then used to train the output weights W^{out}.

Other models, such as a multi-layered perceptron, could have been used for this task. This model was used as it seemed more interesting to gain experience with, as well as the fact that the initial project required the use of an ESN.

B. Initialization

For the model, a reservoir size $L = 200$ was used. The input neuron therefore has L weights ($|W^{in}| = 200$), where each weight w^{in} is sampled from a uniform distribution on $[-\beta, \beta]$. Here, β is the amplifier of the input weights, about which more is explained in the 'hyperparamater tuning' subsection. The reservoir weight matrix W is of size $L \times L$, where each weight is sampled from a uniform distribution on [-1,1]. To classify one input class, there must be one output neuron, hence resulting in 3 output neurons. The output weight matrix W^{out} is of size 10000×3. Both the initial reservoir states X and the output weight matrix W^{out} are sampled from a uniform distribution on [-1,1].

The spectral radius $\varrho(W)$ can be used to scale down the reservoir weight matrix W. This is important as a large W can cause undesirable behavior such as fixed points and chaos (Lukoševičius, 2012). The spectral radius is is the largest absolute eigenvalue $|\lambda|_{max}$ of W. The weight matrix is then scaled by $W * (1/\varrho(W))$. This ensures a spectral radius of 1 and therefore protection against undesirable behavior. To fit it perfectly to the input data, W is scaled again (through multiplication) using a hyperparameter called amp_W. This is discussed further in the hyperparameter section.

C. State computation

Now that the model has been initialized, the reservoir states can be computed. The learning task does not require that the output weights be computed along with the passing of the temporal input (as it would with an RNN). Therefore it is possible to collect all reservoir states before the optimization of W^{out}. For each timestep t, and activation level of a neuron state $x(t)$, the next state for a neuron $x(t+1)$ can be formulated as follows:

$$x(t+1) = (1-\alpha)x(t) + \alpha\sigma(Wx(t) + W^{u}u(t)).$$

Here, α is the leaking rate and $\sigma()$ is the sigmoid function. The leaking rate defines how quickly a neuron state can adapt to new input. A lower α results in a more slowly adapting neuron. For the sigmoid function, the function $tanh()$ was used to produce non-linearity.

Often a bias is used as well, although it was found to be unhelpful in this learning task. The size of the timesteps is set at 0.01. The leaking rate is further discussed in the hyperparameter subsection.

D. Washout point

As the reservoir states are initialized pseudorandomly, and the states of previous iterations will echo on to new states, this randomness will influence the first section of computed states. These states can influence the computation of the output weights and so the effect of the initial random states needs to be removed from the reservoir states. The duration of this effect is called the 'washout' period and it can be found by analyzing a plot of the reservoir states. The washout point is the point at which there is no more trace of the pseudorandomly initiated states in the reservoir. By plotting a number of reservoir neuron activation levels $x(t)$ in 2 it can be observed that the washout point is around 40 iterations. To account for possible outliers, the washout point was set at $T = 1$. This means that only the reservoir states after 100 iterations are used to train the output weights. As the input sample S was of size 2×10000, it is now of size 2 $times9900$. The training data and labels are therefore now of size $|U^{in}| = |Y^{in}| = 9900$.

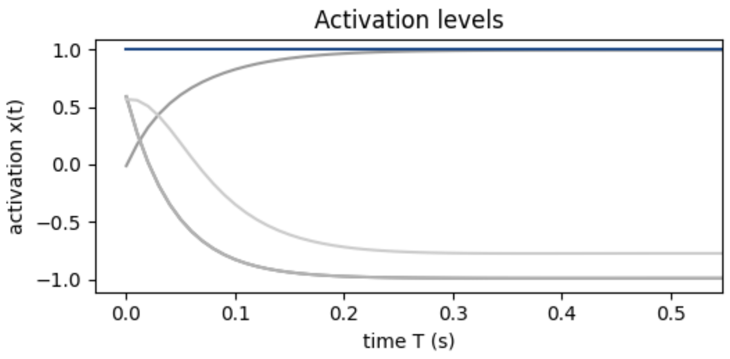

Fig. 2: The pseudorandom initialization at $T = 0$ has an effect on the reservoir neuron states $x(t)$. This effect has completely disappeared around $T = 0.4$.

E. Learning

The learning task of an ESN is to find the optimal output weight matrix W^{out} that produces a minimal loss between the predicted classifications Y^{in} and the output labels Y^{out}. As such, the task can be described as follows:

$$W^{out}_{opt} = argminL(Y^{out}, Y^{in}).$$

Here, $L()$ is the loss function and W^{out}_{opt} is the optimal output weight matrix. For a classification task, categorical cross entropy is often used as loss function. However, this is computationally more expensive than a mean-square error (MSE) loss and so the latter was used in the final algorithm. Calculating the loss L using the mean-square error is formulated as follows:

$$L = \frac{1}{n}\sum_{i=1}^{n}(y^{out}_i - y^{in}_i)^2,$$

where n is the number of samples, y^{in}_i is the sample label, and y^{out}_i is the classification. This results in a score (the loss) that becomes smaller as the difference between y^{out}_i and y^{in}_i becomes smaller. That is: the more similar the output of this model is, the lower value L becomes. Hence why the best output weights are found by minimizing the loss. The learning is done via ridge regression, which is formulated as follows:

$$W^{out} = argmin\frac{1}{N_y}\sum_{i=1}^{N_y}(\sum_{t=0.01}^{T} L(y^{out}_i(t) - y^{in}_i(t)) + \beta \parallel w^{out}_i \parallel^2).$$

Here, N_y is the number of training samples, T is the training period, $L()$ is the previously mentioned loss function, β is the regularization factor, and w^{out}_i is $W^{out}[i]$. The β is used to tune the focus between a small loss and small output weights w^{out}. In this model, β was manually tuned to $1e-5$.

F. Hyperparameter tuning

In order to find this model's optimal hyperparameter settings, a script was executed to test different combinations of α, β, and the amp_W. This process is shown in Algorithm 1 in the appendix. Inspecting the outcome showed that the model performs well using a wide range of hyperparameters. These specific hyperparameters were chosen for tuning by the advice of Lukoševičius (2012).

The tuning of α was done in the range $[0, 1]$, these are the maximum values that alpha can have. Every iteration, α was incremented by 0.1. The tuning of β was done in the range $[0, 4]$ as tuning a model manually after $\beta = 4$ never resulted in better performance. β was incremented by 0.2 each timestep. The tuning of amp_W was done in the range $[0.6, 2]$, as tuning beyond 2 yields chaotic behavior from the reservoir states.

As mentioned earlier, there are many combinations of hyperparameters that perform well (accuracy $\approx 83\%$) on this task. The final hyperparameters are: $\alpha = 0.2$, $\beta = 3.4$, and $amp_W = 1.2$.

G. Testing

In the introduction it was mentioned that the model performance is to be measured as a percentage of accurate classifications. To do this, it will be tested on 30 newly-generated sets of data D of size $|D| = 10000$. The accuracy is averaged across all epochs.

The output values of this model are in the order of $e-2$ to $e-4$, therefore the predicted output label is determined by the value in the prediction matrix Y^{out} with the largest amplitude.

IV. Results

Before doing a quantitative analysis on the model's performance, let us examine the model predictions. To do this, both the model predictions and input labels are plotted numerically as plotting a one-hot encoded array does suit visual analysis well. This means that the one-hot encoding procedure needs to be reversed before plotting. The result can be seen in Figure 3.

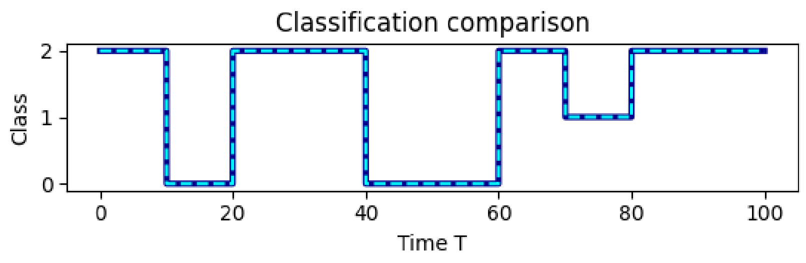

Fig. 3: A comparison between the input labels (dark blue) and and output classifications (dotted, cyan). This classification used the data from Figure 1 as input.

It seems from the plot, that the model is able to perfectly classify the types of waves at each timestep.

As for the quantitative analysis: after computing 30 epochs of accuracy scores, the data was exported in order to visualize the output. The result of this testing procedure can be seen in Figure 4. It can be seen that the accuracy is distributed around 86%. By performing basic analysis, it was determined that the accuracy across 30 epochs was 86.5% with a standard deviation of 1.49%.

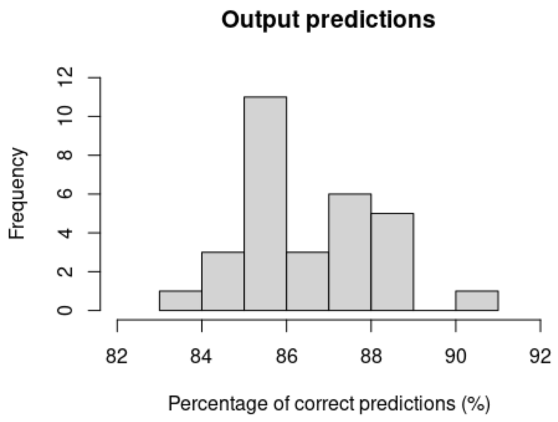

Fig. 4: A histogram showing the frequency of correct prediction percentages

V. Discussion

The aim of this section is to reflect on this project in terms of results, strong points, and improvements. The outcomes of this project will be discussed in $Results$, the general approach in $Reflection$ and a final conclusion will be given in $Conclusion$.

A. Results

While a mean accuracy of 86.5% should be enough to correctly classify a wave function of 1000 timesteps, it does seem like poor performance on a relatively simple task such as this. The main argumentation in the introduction was that both a non-AI algorithm and a NN could perform the same classification task equally as well. This assumed that both were equally adept at producing the classification. So, what went wrong?

A closer inspection of the data shows that nearly all (by visual estimation $\sim$ 90%) of the errors were produced near the input amplitude $u^{in} = 1$. This means that most errors are made at the points where the input values are highest, and change in input is happening most quickly (for the sine and sawtooth functions). To my knowledge, there can be three explanations for this. First, the neurons adapt too slowly to new input, hence the α scaling is wrong (too low). Second, the input does not have a sufficient influence over the neuron states, hence the β scaling is wrong (too low). Third, this behavior is inevitable due to previous neuron states having an influence in the next few iterations.

Furthermore, the classification output generated by the model is small (order of $e-2$ to $e-4$). A possible improvement is to use output feedback or a smaller regularization factor in the learning equation.

Lastly, the size of input data is the same for training as it is for testing. It is only after writing this report that the realization came that the model may need more training data.

B. Reflection

Working on this project has taught me a great deal about both ESNs and working on large projects. When the first features of this project were being implemented, more yet were still misunderstood. Due to the modular nature of ESNs, it can be improved upon by adding new features. For example, features such as the washout period or hyperparameter tuning. This made it possible to work on some improvements for the ESN while others still required further study. It also allows for flexibility in the classification task when a new goal is set. The result is a working product throughout the entire project. Both of these aspects made the project very manageable.

However, these advantages also came with a disadvantage. The pipeline of this project has been changed repeatedly throughout the project, which has slowly become cluttered with traces of previous implementations. The amount of matrix transposes ($mat.T$) used because of this has made the final implementations quite difficult. Therefore a future improvement would be to first define a clear project pipeline before starting on the implementation.

Researching theory at the same time as implementing code has cost a lot of time which could have otherwise been used to work on a dataset. Because of that, the experience gained in this project is only on the side of the learning algorithms. Nevertheless, the learning curve on the side of the learning algorithms has been a steep one. Although the theory was understood quickly, the implementation proved to be more difficult than expected.

Although some libraries were used for computations (numPy and sciPy), all of the methods used in this project were written from scratch (without libraries). Although this has likely resulted in a more inefficient and inaccurate model, it has also strongly helped my understanding of the subject. A goal for the future is to compare self-made methods to ones from libraries in order to improve on the basis of knowledge that has been gained through this project.

C. Conclusion

The goal of this project was to create an ESN that could classify three types of wave functions. Processes such as hyperparameter tuning and initialization washout were implemented in order to improve the model. Although the accuracy of the model was not as high as expected, and a lot of things should be done differently next time, the model did successfully complete this task on a relatively small training set. That, and a lot of newly gained insights into the subject of ESNs make it so that the project can still be considered a success.

References

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., … Oliphant, T. E. (2020, September). Array programming with NumPy. *Nature*, *585*(7825), 357–362. Retrieved from `https://doi.org/10.1038/s41586-020-2649-2` doi: 10.1038/s41586-020-2649-2

Jaeger, H. (2007). Echo state network. *Scholarpedia*, *2*(9), 2330. (revision #196567) doi: 10.4249/scholarpedia.2330

Lukoševičius, M. (2012). A practical guide to applying echo state networks. In G. Montavon, G. B. Orr, & K.-R. Müller (Eds.), *Neural networks: Tricks of the trade: Second edition* (pp. 659–686). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from `https://doi.org/10.1007/978-3-642-35289-8_36` doi: 10.1007/978-3-642-35289-8_36

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., … SciPy 1.0 Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*, 261–272. doi: 10.1038/s41592-019-0686-2

Algorithm 1 Hyper tuning

```
1: for $\alpha = 0.1, 0.2 \ldots, 1$ do
2:     for $\beta = 0.2, 0.4, \ldots, 4$ do
3:         for $amp_W = 0.6, 0.8, \ldots, 2$ do
4:             Generate input samples
5:             Generate a reservoir
6:             Compute reservoir states
7:             Train $W_{out}$
8:             result $\leftarrow$ Test output against labels
9:             if result $>$ best_result then
10:                average_result $\leftarrow$ average(Test against new data 3 times)
11:                if average_result $>$ best_result then
12:                    best_result $\leftarrow$ average_result
13:                    best_hypers $\leftarrow (\alpha, \beta, amp_W)$
14:                end if
15:            end if
16:        end for
17:    end for
18: end for
19: return best_hypers
```