Biologically Inspired Computation Coursework 1

Alexandre Kha and Raphaël Valeri

28th October 2022

Introduction

The objective of this work is the implementation of an Artificial Neural Network (ANN) in Python, without using any Machine Learning library (like sklearn or Tensorflow). Mainly $numpy$ and $matplotlib.pyplot$ are used in this work. The key concepts of this task are the principles of forward propagation, backpropagation and gradient descent, which are crucial in the training process. Their implementation will be detailed in the following parts.

This project would allow a user to design a Multilayer perceptron (MLP) model, by choosing its number of layers, as well as the number of neurons (or nodes) in each, and their activation function. The syntax used to create such ANNs is actually based on the Keras' one, e.g. the user declares an 'empty' model, then adds layers by using the method add_layer (in which the number of nodes and the activation function are specified), compiles the model by specifying the optimizer and the loss function, and ultimately trains the model by using the fit method, where the dataset and the target are notably taken as parameters. The finality of this coursework also consists in testing different hyperparameters (learning rate, batch size, epochs, number of layers ...) in order to find the optimal model configuration in terms of accuracy. In this study, the visualisation of the training curves (accuracy vs epochs, loss vs epochs) can be useful; therefore, our model is also designed to keep traces of the training results.

To evaluate ANN models, breast cancer data issued from `https://archive.ics.uci.edu` will be used. This dataset contains 568 samples with 30 features, and the target is binary, with 211 true positives (malign tumors). K-fold cross validation will then be performed on the model, regarding these data, and the performance evaluation will be assessed by the average accuracy on the K test sets.

1 Program development rationale

In order to design our neural network, we chose the Object Oriented Programming (OOP) paradigm, which has the advantage of defining a clear structure, by regrouping relevant and related values (for example, the hyperparameters) and methods (like fit, $predict$ ...) under a single object. Moreover, OOP benefits from inheritance, which allows to add extensions to the code rather than modify it, which can be particularly burdening.

Therefore, numerous auxiliary concepts, have been implemented in classes, among which : $ActivationFunction$ which defines the possible activation functions (Sigmoid, Tanh, Relu and Softmax) and their derivatives, $Loss$ for the different losses available (MSE, AbsE and

cross-entropy), $Optimizer$ for the descent gradient optimisation (SGD, Batch, Minibatch, RMSPROP and ADAM), and finally $MinMax$ for the feature rescaling.

The most important class of this project is MLP, whose main functionalities are the add_layer method, which enables a user to design each layer of the MLP (taking into account the number of nodes and the activation); as well as the fit method, which trains the model by gradient descent , once the dataset, the target, and some hyperparameters (learning rate, batch size and number of epochs), have been specified. The batch size parameter is ignored if the optimizer chosen in the $compile$ method is "sgd" or "batch".

2 Methods

2.1 Programming formalism

Once all the weights (and biases) have been randomly initialised, we need to update them after applying gradient descent, which is the most technical part of the project.

We consider, without loss of generality, the case of minibatch descent, where the training set is split in minibatches of a certain size (chosen by the user). During an epoch, the gradient descent is applied on each of these minibatches. The gradient descent process can be split into three distinct functions : forward propagation, backpropagation and weights update.

Assuming we have L layers, forward propagation can be expressed for $n \in \{1, ..., L\}$, according to our code formalism, by :

$$Out[n] = Activation[n].value\left(Weights[n] \times Out[n-1] + Bias[n]\right)$$

Note that $Out[n]$ is the n^{th} layer's output and $Out[0]$ corresponds to the input data.

Then, backpropagation computes the loss derivatives by exploring the neural network backward (from last layer to first), which, on the output layer, can be expressed by using the chain rule (y being the minibatch target) :

$$dZ[L] = \partial_Z Loss(y, Out[L]) = \partial_a Loss(y, a).\partial_Z\left[a\right] = error.derivative(y, a).Activation[L].derivative(a)$$

However, in the multiple outputs case, this calculation (even without the chain rule) does have a very handy simplification when using Softmax activation and Categorical Cross Entropy loss (CE). Indeed, with m the number of classes, and y a one-hot encoded target ($y_i = 1$ if in class i, else $= 0$), we have by differentiating the CE loss in respect to Z_k :

$$\partial_{Z_k} CE(y, a) = -\sum_{i=1}^{m} \partial_{Z_k}\left[y_i log(a_i)\right] = -\sum_{i=1}^{m}\left(\frac{y_i}{a_i}\partial_{Z_k}\left[softmax(Z_i)\right]\right) = -\sum_{i=1}^{m}\left(\frac{y_i}{a_i}\partial_{Z_k}\frac{e^{Z_i}}{\sum_{j=1}^{m} e^{Z_j}}\right)$$

By making distinction between $i = k$ and other i values, we ultimately obtain :

$$\partial_{Z_k} CE = \left(\sum_{i=1,\, i \neq k}^{m} y_i a_k \right) - y_k + y_k a_k = a_k \left(\sum_{i=1}^{m} y_i \right) - y_k \underset{one\ hot}{=} a_k - y_k$$

$$\implies dZ[L] = \partial_Z CE(y, Out[L]) = Out[L] - y$$

The last steps of the backpropagation, which consist in computing $dW[n]$ and $dB[n]$ from $dZ[L]$ and $dZ[n]$, are not detailed, but do not raise any particular difficulty.

The third and last phase of the gradient descent is the updating of the weights. In the most standard case, the new parameters can be written as (with γ the learning rate) :

$$Weights[n] := Weights[n] - \gamma\, dW[n] \ \ , \ \ Bias[n] := Bias[n] - \gamma\, dB[n]$$

Our project does also implement the Root Mean Square ($RMSPROP$) and the Adaptive Moment Estimator ($ADAM$) optimizers, which give a more complex form to the gradient descent, but enable a very fast gradient convergence. [2] In $ADAM$ case, for $n \in \{1, ..., L\}$, the final gradient descent form can be expressed by :

$$Weights[n] := Weights[n] - \gamma\, \frac{V_w^{corr}[n]}{\sqrt{S_w^{corr}[n] + \epsilon}} \ \ , \ \ Bias[n] := Bias[n] - \gamma\, \frac{V_b^{corr}[n]}{\sqrt{S_b^{corr}[n] + \epsilon}}$$

Where $V_w^{corr}[n]$, $V_b^{corr}[n]$, $S_w^{corr}[n]$ and $S_b^{corr}[n]$ are the (n^{th} layer) bias-corrected 1^{st} and 2^{nd} moments, which are updated as well at each gradient iteration.

2.2 Method of hyperparameters tuning

In order to define the most relevant ANN to solve our classification problem, we need to investigate the effect of the hyperparameters. First, we need to set the number of hidden layers, the number of neurons and the activation function for each layer. The training process requires also some hyperparameters like the loss function, the learning rate, the batch size and the number of epochs. For each of these hyperparameters we determined a range of possible values and a 10-fold cross-validation has been performed to compare the accuracy and the training time related to each value.

3 Results

3.1 Results of hyperparameters tuning

The results of the investigation for the number of hidden layers is shown in figure 1. We can see that the accuracy decreases when the number of layers is greater than 2 and that the dispersion of the accuracy values on the different folds increases as the time spent to fit the

model. We also observed that the ReLu activation function seems to have greater perfor-

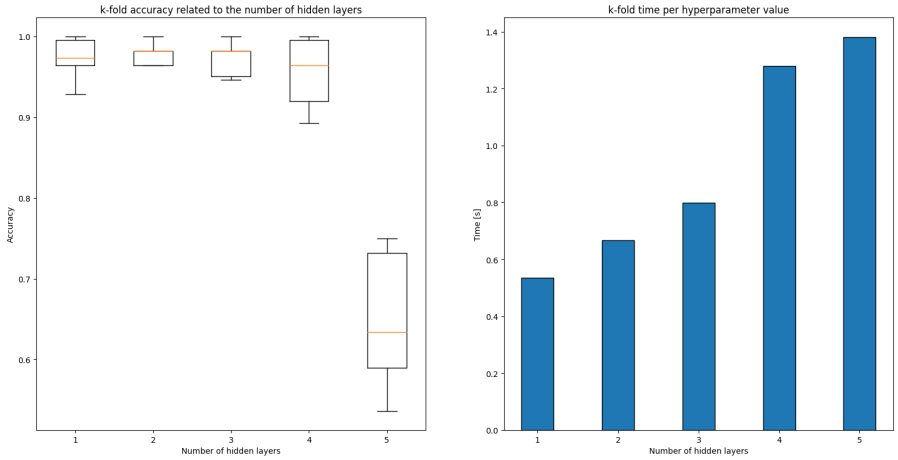

Figure 1: Hidden layers tuning

mances (accuracy value and dispersion, and time) than the hyperbolic tangent. Therefore, the optimal ANN architecture appears to have **two hidden layers with 64 and 32 neurons and ReLu activation**.

This configuration has been used to test the other hyperparameters. The most influential hyperparameters seem to be the architecture of the output layer, the batch size and the learning rate. We did the same investigations as previously and found that **the output layer with 2 neurons and softmax activation function** is quite better than with 1 output neuron and sigmoid or hyperbolic tangent activation function. The results of batch size investigations are presented in figure 2 with a learning rate set to 0.01, which appears to be the optimal value.

We can see that, as it is expected, the best trade-off between the number of the batch-size is between, a low batch-size, which leads to more updates of the weights but which can be more affected by noise and need more computational time; and a huge batch-size, which decreases the number of updates of the weights as well as the influence of each particular samples in the update. In our case it appears that **a batch size of 8** seems to be optimal. As we are using softmax and 2 output neurons, the most relevant loss function seems to be the **cross-entropy** compared to Mean Square Error (which is by the way, more suited to regression problems).

3.2 Training and test of the model

With the hyperparameters set at the values mentioned in the previous paragraph, the training curve of the accuracy and the loss of the model is presented in the figure 3. We can see

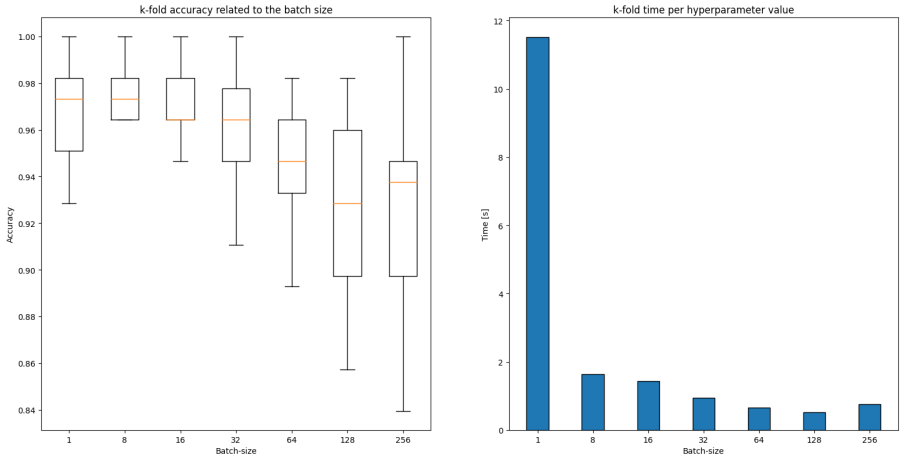

Figure 2: Batch size tuning

that the convergence of the model is fast and high with the training data as well as with the validation data. This models gives an accuracy of 0.9804 ± 0.0096 with 10-fold cross-validation.

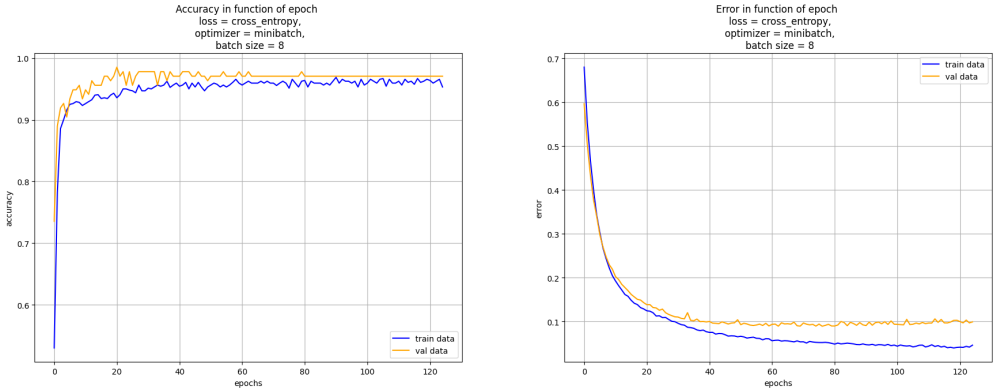

Figure 3: Training curve of the accuracy and the loss of the model

4 Discussion

As previously seen in the results section (3), the model performances are very sensitive to the hyperparameters values like the architecture of the MLP (number of hidden layers, number

of nodes and activation function of the layers, output layer) and the training hyperparameters like the batch-size, the learning rate or the number of epochs (which could lead to an overfitting problem if it is too high). Our choices of the potential values of hyperparameters were influenced by the literature. For example, the use of ReLu activation function in hidden layers and softmax (or sigmoid) in the output layer seems to be very popular and relevant. [3] Our implementation of the Adam and RMSProp algorithm (from [2]) seems to be useful with a lower value of the learning rate (typically 0.001) but the hyperparameters investigation showed that the value of 0.01 was more useful to compare the other hyperparameters. Even if these optimizations have not been used here, they could probably be useful with other datasets, as well as the batch normalization. [1]

Conclusion

To conclude, this implemented Multilayer Perceptron in Python allows the user to customize his own model with the most basic and required hyperparameters, but also with more advanced optimization hyperparameters, as it is possible to use Adam and RMSProp algorithms as well as batch normalization. These additional features would make this program suitable to more challenging datasets.

Our method relies mainly on using OOP's handy features (classes, attributes, methods and inheritance) as well as storing the layers' weights and bias in a dictionary, which enables to easily update them and connect the layers.

Finally, the hyperparameters' investigation gives an overview of the possible values of the parameters in this problem. The parameters found allow to reach an accuracy of 0.98, which demonstrates a good performance of our model.

References

[1] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[2] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017.

[3] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *CoRR*, abs/1811.03378, 2018.