Group 2215 - Deep Neural Network

Agosti Luca, Ballout Fatima, Bee Nicola, and Saccaro Lorenzo

(Dated: 19/03/2022)

Deep learning neural networks produce excellent results in various pattern recognition tasks[1]. Such remarkable performance can only be achieved by carefully studying how to convert input data into meaningful knowledge at output. This is done, first and foremost, by finding the optimal combination of hyper-parameters for the DNN. In this work, we systematically assess the optimization of such parameters for the training of a neural network, given a simply classification task. In particular we aim to correctly identify some points belonging to two different regions of the 2D plane. Furthermore, we also present an analysis on how the size and augmentation of the input dataset can influence the overall performance of the DNN.

INTRODUCTION

The purpose of this paper is to build and train a deep learning neural network (DNN) to solve a basic binary classification task. A fully-connected DNN is composed by a series of layers of neurons, each one applying a nonlinear function, called *activation function*, to the weighted sum of all the outputs of the previous layer. The first layer, called *input layer*, is where the information is defined. The following ones are the *hidden layers*, where each node is called a hidden node. The last layer is the *output layer*, which is directly related to the target value that the model is trying to predict.

The parameters that describe the architecture (number of layers, neurons per layer, etc..) and other aspects of the neural network are called the *hyper-parameters* (HP). Finding the optimal HP is essential to achieve better accuracy and will be a main topic in this work. By saying that the "neural network is learning" we refer to the minimization of the *loss function*. This function is an estimator of how much the predicted labels reflects the correct ones and, in our case, is the binary cross-entropy defined as follows:

$$C = \sum_{i \in B} \left[-y_i \ln \hat{y}_i - (1 - y_i) \ln(1 - \hat{y}_i) \right]$$

where B is the set of data and, for each datum i, y_i is the correct label while $\hat{y}_i$ is the predicted one. The minimisation of the loss function is done using a gradient-based algorithm called *optimizer*, like SGD or ADAM, that updates the values of the weights of the network accordingly.

The particular task assigned to our neural network is the classification of some points belonging to two different regions of the 2D plane. Considering a square box of 100×100 centered in the origin and a triangle with the vertices in $(-20, 60), (80, -40), (-20, -40)$ (we identify this shape as `type 1`), we label with "1" the points inside the triangle and with "0" the points outside it. The dataset is generated using a 2D-uniform random function with boundaries given by the shape of the box, as shown in Fig. 3.

METHODS

We generate N points for the training set and another 10% for the test set. No validation set is specified since the grid search function implemented in the scikit-learn package [2] performs a cross validation: the train set is split in k subsets each of size N/k. Each subset is then used once as validation while the other $k-1$ are used for training. In this way, starting from the same training set, we can obtain k validation scores that can be averaged to have some minimal statistical value. In this work k is set equal to 5. Before the data are fed to the network they are rescaled to be in the interval $[-1, 1]$ by dividing the coordinates of each point by 50. We then define a function that, using Keras [3] with Tensorflow [4] backend, builds a neural network according to all the different HP, passed as input.

The HP taken into account are the following:

- **Batch size:** number of samples processed before the model is updated. The size of a batch must be in the range $[1, N]$ where N is the number of samples in the training dataset.

- **Number of hidden layers** By adding more hidden layer the depth of the network is changed, but the complexity is increased, too.

- **Number of neurons:** how many neurons there are in each hidden layer. The input layer has a number of neurons equal to the dimension of the input (2 in our case) while the output for a binary classifier is a single number (single neuron).

- **Activation function:** it is a nonlinear function applied to each neuron before passing the value to the next layer. It compute its output taking in input the sum of all the weighted values of the previous neuron. For the output layer this is fixed to the sigmoid function that provides an output in the $[0, 1]$ range.

- **Optimizer function:** is the function selected to find the best minimum of the loss function. Since the function to be minimized often has many different local minima, a particular choice of the optimizer function can not only lead to faster convergence, but to dissimilar results, too.

- **Learning rate:** it is a parameter used by the optimizer function to determine how large are the steps. Bigger steps makes the optimization faster but can lead both to avoid small local minima and to divergence, thus a compromise is required.

- **Dropout rate:** Dropout is a technique that temporarily removes random neurons during the training of the neural network helping with the overfitting issue. We apply dropout only to the last hidden layer.

- **Weights initialization:** is a procedure that assigns the initial weight values to all connections between neurons according to a selected function.

The brute-force way to find the best combination of HP (the one with the highest validation score) would be by doing a grid search between all of them. For example, if we would like to tune the learning rate trying the values [0.1, 0.01] and the optimization function trying [`'SGD'`, `'ADAM'`] we would end up having four different pairs of HP to test. Following this procedure for every possible interval of every HP, should allow us to determine exactly what is the best combination of HP for our problem. Unfortunately this is usually not possible (unless the amount of all possible combinations of HP is very small) due to computational power and time constraints. One could try to optimize each HP individually [5] but that is not ideal since most of them are correlated to each other. However, we follow a different approach from those two: a way that allows us to use the granularity granted by the grid search without having the unbearable computational time problem. We perform 3 smaller grid searches rather than only a global one, grouping together HP that are more dependent on each other: this greatly reduces the number of combinations to test. The downside of this method is that performance evaluation has to be done by fixing the HP which are not being evaluated in a particular search, and this could affect the score of that specific combination. We mitigate this problem by keeping not only the best combination of each search, but we use the top 5 in the following ones. The first grid search is between batch size, optimizer and learning rate. The batch size is usually fixed to the largest value that fits in memory (GPU or system) and it is often a power of 2. In our case we do not suffer for memory constraints so we tested common values such as 32, 64 etc. The learning rate tested are $[0.0001, 0.001, 0.01, 0.1, 1]$, while the optimizers used are the most common ones (see the attached notebook for the complete list of HP tested). Note that we do not include the number of epochs in our HP search: we fix it to a large number (1000) and then we use the early stopping callback to halt the training when the accuracy does not improve through a certain number of epochs, e.g. 10 or 50. The second grid search is for the architecture of the DNN, i.e. number of layers, number of neurons and activation function. The last one is for the fine-tuning parameters: dropout and weight initializers. After each grid search is performed the results are saved in a text file. Lastly, after all three steps, the best-score model is then trained using the entire train test and its performance checked against the unseen test set.

This procedure is then repeated varying the size of the training set to study its effect on the model accuracy: the considered N values are 800, 4000, 8000 and 20000. Moreover, we generate a training set by augmenting previous generated data to see if this would help to achieve better results in a scenario when it is impossible (or difficult) to obtain more data. In our case we do it by adding to the coordinates of each one of the input points a small random offset obtained from a Gaussian distribution with $\sigma = 0.2$: starting from $N = 4000$ we obtain a training set of 8000 points (labelled as 4000 + aug).

One last test we would like to perform is to observe the behaviour of our DNN when trained to solve a different task. To do that we define a new classification problem (`type 2`): inside a square 100×100 centered in the origins we label with "1" and "0" respectively the points that satisfy or do not satisfy the following condition:

$$\text{sign}(x+y)\ \text{sign}(x)\ \cos\left(\frac{\sqrt{x^2+y^2}}{2\pi}\right) > 0$$

This new regions can be seen in figure 4. We tried both to train our DNN with the best HP found for the previous problem (with $N = 4000$) and to repeat the HP optimization from scratch using the same grid searches as discussed above.

RESULTS

We show the best HP combination for each configuration (N and `type`) in Tab. I. The preferred activation function is the *LeakyReLU* that differs from the standard *ReLU* because it allows a small, positive gradient when the unit is not active. Regarding the optimizer and the learning rate Adam and 0.001 are the most common combination. We can observe that the `type 1` classifiers tend to have simple architectures with a total number of hidden neurons between 20 and 60. We come back later to discuss the `type 2` case. The dropout rate has a bigger role when N is larger in particular when testing the augmented set, while for the smaller training samples it is always below 20%. Finally, the glorot initializer (both normal and uniform) overcomes the standard ones

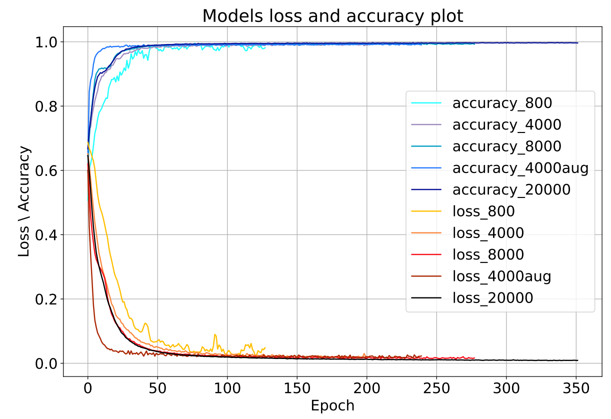


Fig. 1: Trend of the loss and accuracy metrics of the tested models during training as a function of the elapsed epochs

by truncating the respective distributions based on the number of input and output units.

We now discuss how different values of N impact performance and other aspects of our model. First of all, in Fig. 1 we plot the loss and accuracy during the final training for every N. We observe that larger training sets require more epochs for the optimizer to converge: more iterations are needed to extract all the information stored in the data. Moreover we can see the effect of the lower learning rate for $N = 800$ which causes more fluctuations during the minimization. The augmented set displays a steeper descent than the other ones even if the learning rate and optimizer are similar to the $N = 8000$ and $N = 20000$ sets.

In Fig. 2 we plot the average validation accuracy over the $k = 5$ folds (the error bars are the corresponding standard deviations) and the test score for each N tested for the `type 1` function. As expected increasing the training set size benefits the performance of the model with an increase in accuracy of both validation and test. However this comes with the penalty of longer training times, also plotted in Fig. 2. A compromise between performance and computational time can be achieved: if we look at the $N = 8000$ and $N = 20000$ sets an improvement of only +0.2% of the accuracy value comes at the cost of an additional 160% training time (usually not worth it). We can observe the absence of overfitting: the test and validation accuracy are always compatible with each other. The standard deviation of the validation accuracy decreases with N: this is due to the fact that each fold is more sensitive to the specific split between train and validation set when the number of generated point is lower.

To better visualize what the model thinks is the distribution of our data we generate points to span the entire grid and we plot (Fig. 3) the true label according to the `type 1` function, the model prediction value (a number in the $[0, 1]$ range) and the corresponding output label.

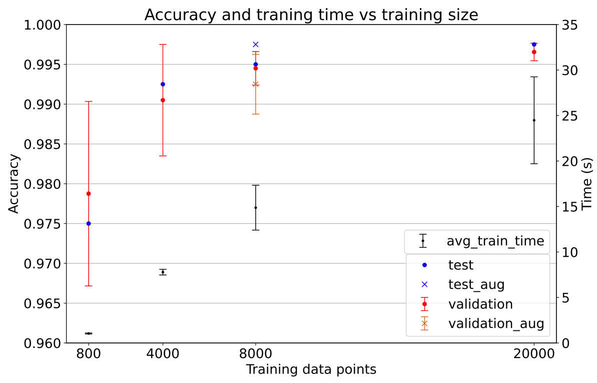
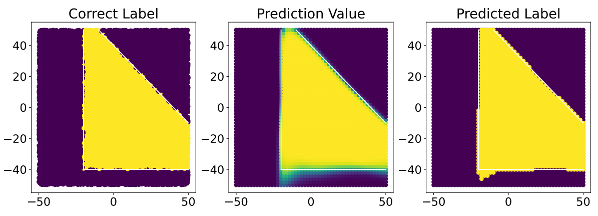
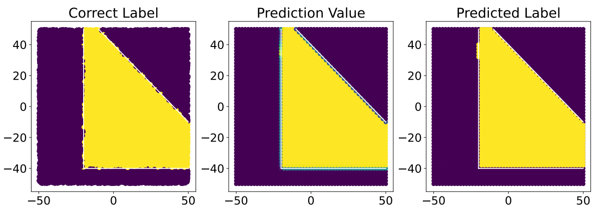


Fig. 2: Test and average cross validation accuracy with corresponding standard deviation, with average training time as a function of training set size



(a) $N = 800$



(b) $N = 20000$

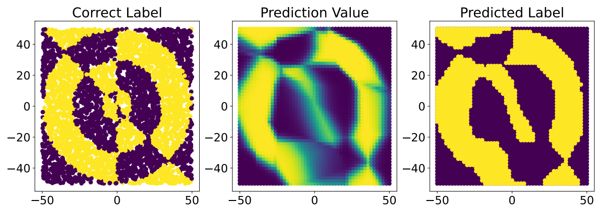
Fig. 3: Main classification task (`type 1` function). From left to right: correct labeled points, predicted values (green shaded points assume values between 0 and 1), and predicted labels

It is easy to spot that the $N = 800$ model has trouble classifying correctly the points along the edges of the triangle especially near the vertices, while with $N = 20000$ the model has an accuracy of 99.7% and the shape reconstructed is basically perfect.

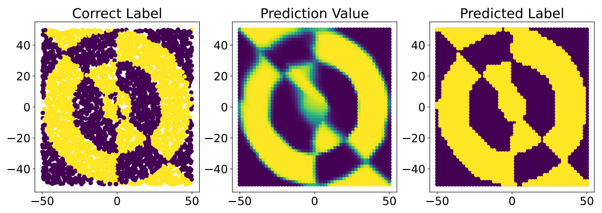
Moving on to the `type 2` classification problem, the same visualization plot is shown in Fig. 4. When we train the best model found for the `type 1` function (with $N = 4000$) with the new generated data we obtain unsatisfactory results. The accuracy reaches only 92.0% and we can see that the model struggles to understand what happens in the center and in the lower part of the grid. If, instead, we perform the HP search from scratch we obtain better results: the accuracy is now up to 97.3% and the predicted shape is more accurate too, even if there are still some problems in the very center part of

	Training set size (N)					
	`type 1`					`type 2`
	800	4000	8000	4000 + aug	20000	4000
Hidden layers	3	2	2	3	2	3
Neurons	10	20	10	20	20	100
Activation function	elu	LeakyReLU	LeakyReLU	LeakyReLU	LeakyReLU	LeakyReLU
Learning rate	0.01	0.0001	0.001	0.001	0.001	0.001
Batch size	128	32	32	32	64	128
Optimizer	Adam	RMS prop	Adam	Adam	Adamax	Adam
Weight initializer	glorot normal	glorot normal	glorot uniform	glorot uniform	glorot normal	glorot normal
Dropout rate	0	0.2	0.1	0.5	0.4	0.1

Tab. I: Best hyperparameters combination found for each training set size (N) and for both `types`



(a) with previously found HP



(b) with tuned HP

Fig. 4: `type 2` function used to check the adaptability of our model ($N = 4000$). From left to right: correct labeled points, predicted values (green shaded points assume values between 0 and 1), and predicted labels

the grid. To further improve the performance more data are needed to have better granularity but a more complex architecture could be required too. Indeed we notice that the optimal combination of HP uses the highest number of total neuron that we test for (300).

CONCLUSIONS

From this work we can take some key aspects on the binary classification problem with DNN. The first is that simple problems can usually be solved with simple (in terms of architecture complexity) models, while more complicated patterns require deeper networks. Increasing the data sample helps achieving better performance but it slows the training process. After a certain threshold a diminishing return effect kicks in: using more data has minimal impact on the accuracy while still increasing the required time. Data augmentation is useful when needing more training data: in our case it does not help in achieving better accuracy but the loss function has a faster convergence. This should be investigated further, e.g. implementing augmentation when using smaller sets, with more complex classification tasks and trying different approaches other than adding Gaussian noise.

Tuning the HP with the scikit-learn grid search is very slow, due to its combinatorial approach, and inefficient: if a specific HP has a value that greatly reduces the performance of the model we still test all the combinations that contain that value. Solutions to this problem have already been found, methods like random grid search or Bayesian optimization can help tuning the HP faster and more efficiently.

Finally the model with the HP tuned for the `type 1` classification problem fails when we try to use it for a different, more complex, task. This is not surprising since it is not designed to be generalizable, but only to deal with a specific problem. More complex models and advanced techniques are needed to achieve a higher flexibility.

[1] P. Mehta, M. Bukov, C.-H. Wang, A. G. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, Physics Reports **810**, 1–124 (2019).
[2] F. Pedregosa, G. Varoquaux, *et al.*, Journal of Machine Learning Research **12**, 2825 (2011).
[3] F. Chollet *et al.*, "Keras," (2015).
[4] M. Abadi, A. Agarwal, *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," (2015), software available from tensorflow.org.
[5] J. Brownlee, "How to grid search hyperparameters for deep learning models in python with keras," (2016).