Deep Neural Networks for Non-Linear Classification Tasks

GROUP 2221 - Alessandro Maria Capodaglio, Sarvenaz Babakhani, Reza Rajaee, and Elham Khosravi
(Dated: March 18, 2022)

When it comes to complicated non-linear classification tasks, Deep neural networks have been shown to be a very viable and efficient option; however proper care must be taken in selecting the structure of the model, i.e. its hyper-parameters. The goal of this paper is to study the performance of the model proposed during the lectures and find a better model which performs sufficiently well on datasets generated by different non-linear functions.

INTRODUCTION

Our first topic of interest was to see how the original, unchanged model would perform if the data available were decreased, increased, or artificially augumented. Subsequently we probed into finding a better model structure, i.e. a neural network with different hyper-parameters, by using a grid search technique. The best model structure found was then compared to the original model on various datasets, with satisfying results.

The first function we tried to teach the model is the one seen during the lectures, which assigns a label to a given point (x_1, x_2) according to the following criteria:

$$f(x_1, x_2) = \begin{cases} 1 & \text{if } (x_1 > -20, x_2 < -40, x_1 + x_2 < 40) \\ 0 & \text{otherwise} \end{cases}$$

METHODS

In order to study how well the original network, described in figure (1) could learn when supplied with varying amounts of data, we decided to plot the training loss and validation loss for 20 different sizes of inputted dataset. The sizes studied were:

- $N = [4000, 3200, 2800, ..., 1200, 800, 400]$
- $N = [4000, 8000, 12000, ..., 32000, 36000, 40000]$

One important thing to note is that when increasing or reducing the size of the dataset, the validation set size was varied proportionally. In other words the ratio of `training_set_size` to `dataset_size` was always 0.8 and the ratio of ratio of `validation_set_size` to `dataset_size` was always 0.2.
When *augumenting* the data, i.e. artificially creating "new" data starting from the original dataset of 4000 instances, the validation set size was kept of size 800 throughout.

As previously mentioned the structure of the model was kept unaltered for this part, the description of the model is shown in figure (1).

We then studied the performance of the DNN when trained on datasets of different sizes which were generated starting from the 4000 original points.

```
model = Sequential()
model.add(Dense(L, input_shape=(L,), activation = 'relu'))
model.add(Dense(20, activation = 'relu'))
model.add(Dense(20, activation = 'relu'))
model.add(Dropout(0.2))
model.add(Dense(1, activation = 'sigmoid'))
nepoch = 400
```

FIG. 1. The model which was trained on different training set sizes. The parameter `L` refers to the number of features of the input, which is always equal to 2.

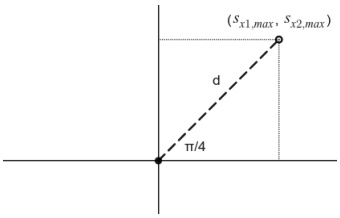

FIG. 2. The origin represents a given point x_i, the distance to its closest neighbor is d. The value of the maximum steps allowed in the x_1 and the x_2 direction are given by $d \cdot \frac{\sqrt{2}}{2}$.

The way these new datasets were artificially created was by applying a *small* shift (s_{1_i}, s_{2_i}) to each point $x_i = (x_{1_i}, x_{2_i})$ thereby obtaining $x_{i_{new}} = (x_{1_i} + s_{1_i}, x_{2_i} + s_{2_i})$. In ordere to accomplish this we had to define a notion of "small" shift. Since the underlying distribution of the data isn't known (in a real world setting) what could be a small shift for one instance may not be a small shift for another instance of the dataset. What we have done can be surmised in the following way:

- for each point in the real dataset calculate the distance d to its closest neighbor. These distances were stored in an array in which the i_{th} element of the array corresponds to the distance that the i_{th} point has to its closest neighbor;
- for each one of these distances we defined a *maximum* step in both the x_1 and x_2 direction. We did this by considering the *worst* case in which the line connecting x_i to its closest neighbor has an angle of $\frac{\pi}{4}$ radians with respect to each of the feature directions. These maximum steps $(s_{x1,max}, s_{x2,max})$ can be thought as the values of the shift for which one point becomes another one. Figure (2) shows a sketch of the procedure thus far.
- create one (or more) copies of the dataset. These

copies were obtained by shifting each feature of each point by a quantity that is less than or equal to the maximum step allowed (for that point). The labeling of the shifted dataset was the same as the original one.

RESULTS

In order to present the results of our labour it's best to give a quick recap on *epochs* in regards to neural networks. An epoch is when the entire dataset is passed forwards (to calculate the outputs of each node in each layer) and backwards (to calculate the parameters of the model through backpropagation) through the neural network exactly once. Concisely speaking, the more epochs the model lives through, the more the parameters are adjusted according to the training set. Clearly too many epochs may, and frequently do, lead to overfitting. The concept of overfitting is among the most studied in the theory of learning. One universally adopted practice to reduce overfitting is to define a stopping point. What we have defined as the best epoch satisfies two requirements simultaneously:

1. it has the lowest training loss
2. the validation loss does not exceed the training loss

In order to be perfectly clear regarding our stopping criteria table [I] illustrates it with an example using actual data from our work.

Epoch	Train_loss	Validation_loss
135	0.0299	0.0241
142	0.0290	0.0262
400	0.0230	0.0374

TABLE I. This data was taken from training the model on 28800 training samples (augumented).
Epoch 135 has the lowest validation loss, but doesn't perform as well on the training data as the other two epochs. Epoch 400 has the lowest training loss but the validation loss exceeds it. Our code chooses epoch 142 as the best epoch, i.e. the stopping point.

Reducing, Increasing, Augumenting

Figure (3) shows the results obtained when training the model on an increasingly smaller dataset. A definite trend in both the training loss and validation loss can be observed. As the size of the inputted dataset decreases both the training loss and validation loss increase, this is a symptom of underfitting. Furthermore, throughout the epochs the validation loss fluctuates a lot. As could've been expected the model performs very poorly.

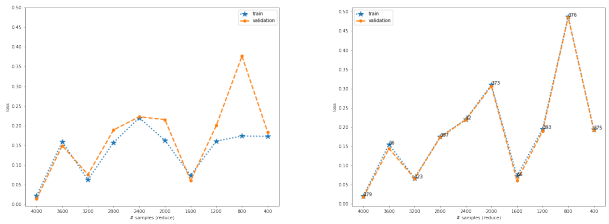

FIG. 3. Panel (a) shows the training loss and validation loss of the 400th epoch, while panel (b) shows the result of the best epoch, which was selected using the procedure described at length before. Note that the number of samples on which the model was trained on is decreasing left to right.

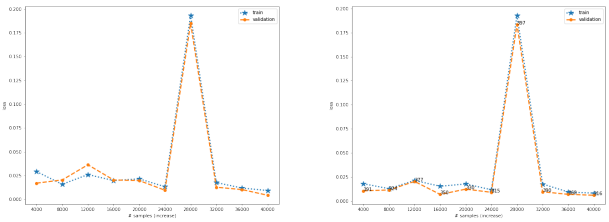

FIG. 5. Panel (a) shows the training loss and validation loss of the 400th epoch, while panel (b) shows the result of the best epoch. Note the different scale of the loss axis with respect to figure (3).

Figure (5) shows the results obtained when training the model on an increasingly larger dataset. As before the loss curves at the last epoch are very similar to the loss curves at the best epoch. It's interesting to observe that, unlike the reducing case, the best epoch is always towards the end of the training procedure. This is most likely due to the fact that, having access to more data, the model benefits from living through more epochs (i.e. going through the data more times).

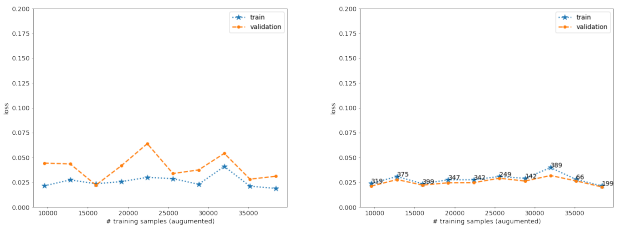

FIG. 4. Panel (a) shows the training loss and validation loss of the 400th epoch, while panel (b) shows the result of the best epoch. Note that the scale of the loss axis is the same as the one in figure (5).

Figure (4) shows the results obtained when training the model on an artificially augumented dataset of various sizes. As before the loss curves at the last epoch are very

similar to the loss curves at the best epoch.

By comparing figures (5) and (4) there aren't any particularly big differences, this means that the augumentation procedure which we have implemented was particularly successful.

Grid Search

Figure (8) Panel (a) shows the learning curve of the model when trained on the unaltered dataset; it can be seen that the neural network is underfitting the data [1][2] and has the capacity to learn more. We were interested into finding a model with better performances. In order to do so we decided to implement a grid search procedure [3] on possible hyperparameters. Initially we also used the random search [4] (`RandomizedSearchCV()`) that takes less time than the standard `GridSearchCV()`, however we found that the latter found a better performing model for all inputted data, so we procedeed with that. Figure (6) shows how the code describing the neural network was changed in order to allow a grid search procedure to be applied to it.

```
def create_model(activation='relu', optimizer='adam', n_layers=4, neurons=20, dropout_rate=0.2):
    model = Sequential()
    model.add(Dense(2, input_shape=(2,), activation='relu'))
    for i in range(n_layers):
        model.add(Dense(neurons, activation=activation))
    model.add(Dropout(dropout_rate))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model
```

FIG. 6. The neural network which will be outputted by the grid search will have an input layer which remains unchanged, followed by a certain `number of layers` each containing a `number of neurons` with the same `activation function`, the last of said layers will have a `dropout_rate`. The actual training of the model will also be done by a specific `optimizer`. All the parameters written in `typewriter font` are tuned by grid searching among a range of possible values.

The result of the grid search provided us with a model structure described in figure (7), which was then tested on the original dataset.

```
model = Sequential()
model.add(Dense(L,input_shape=(L,),activation='relu'))
model.add(Dense(25,activation='elu'))
model.add(Dense(25,activation='elu'))
model.add(Dense(25,activation='elu'))
model.add(Dense(25,activation='elu'))
model.add(Dense(25,activation='elu'))
model.add(Dense(25,activation='elu'))
model.add(Dropout(0.2))
model.add(Dense(1,activation='sigmoid'))
```

FIG. 7. The neural network structure returned by performing grid search.

Running the data through this new neural network we obtain the learning curves shown in panel (b) of figure (8). By comparing figures (8) Panel (a) and Panel (b) (note the different scale of the loss axis in each plot) it can be seen that the grid search procedure has indeed given us a constructed network which performs better on the original data. Subsequently we wondered if by performing a grid search on different hyper parameters we could get a better performing model. Thus we took the best performing model so far and we performed a grid search also on the activation function of the input layer, batch size and on the epochs, the results of the outputted model on the dataset are shown in Panel (c) of figure (8). Even though the results obtained are satisfactory, by further looking at figure [8] it's apparent that there are severe fluctuations in the validation loss of the best model (Panel (c)) across its epochs. This phenomenon could be due to several things, one possible reason is that we are trying to train a relatively large network with few data points. Panel (a) of figure (9) shows the learning curve for the best model trained on 8000 training samples; we can see that the fluctuations reduce significantly.

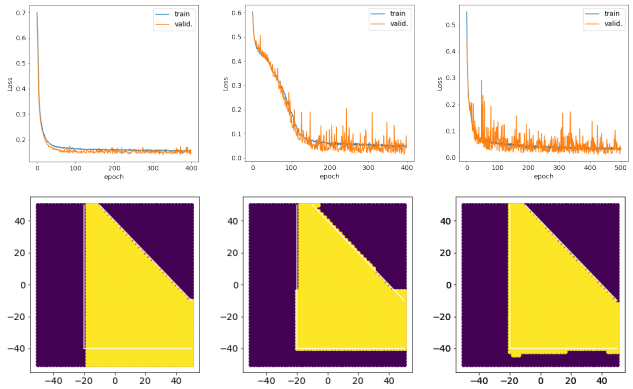

FIG. 8. Panel (a) shows the learning curve for the original neural network, i.e. structured according to the code in figure (1). Panel (b) shows learning curve for the neural network after the first grid search (structured according to the code in figure (7)). Panel (c) shows the learning curve for the neural network after the second grid search. Each panel $(a), (b), (c)$ has the corresponding results on the test set shown below it. Again it is important to note that each of the models was trained on the same 3200 data points.

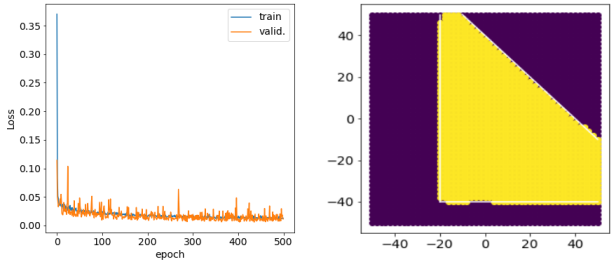

FIG. 9. (a) The training loss and validation loss of the best model found using the second grid search when supplied with 8000 training points. (b) The results of the trained model on the test set.

Another Function

We then compared the learning capabilities of the models on another nonlinear function. We applied the original model and our best model found using grid-search on a dataset generated by the function f_2 which labels a given point (x_1, x_2) according to the following criteria:

$$f_2(x_1, x_2) = \begin{cases} 1 & \text{if } \left(sign(x_1 + x_2) sign(x_1) \cos \frac{||x||}{2\pi} > 0 \right) \\ 0 & \text{otherwise} \end{cases}$$

Table [II] shows the final (at the end of all the epochs) training loss and validation loss of both models trying to learn the labeling function f_2:

Model	Train_loss	Validation_loss
Old	0.478	0.495
New	0.119	0.234

TABLE II. The previous un-optimized neural network fails to learn the labeling function, it is almost as bad as a random two-sided coin toss.

Even though the results are promising for the new model, it could be argued that it is at the brink of overfitting. Figure (10) shows the results on the test set for both models.

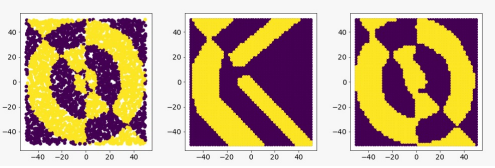

FIG. 10. Panel (a) shows the actual training data, generated by the labeling function f_2. Panel (b) shows the results obtained by the original, un-optimized neural network. Panel (c) shows the results on the test set of the model found using grid search. It is important to note once again that panels (b) and (c) are obtained from training the model(s) on the same 3200 training samples.

Rescaling

All the previous parts of the paper were done by standardizing (which we will call `std`) each feature of the data, i.e. subtracting the mean and dividing by the standard deviation. Here we compare the result of the model for the mentioned rescaling and two others. The other rescaling procedures which were examined are the one discussed during class, i.e. normalizing the data (`norm`), and minmaxscaler (`minmax`). Table (III) shows two random runs of training and validation losses for the original model trying to learn the initial function when trained on 3200 samples which have been rescaled in different ways.

	first_run		second_run	
Re-scale	Train_loss	Validation_loss	Train_loss	Validation_loss
`std`	0.165	0.155	0.050	0.043
`norm`	0.056	0.029	0.011	0.010
`minmax`	0.029	0.017	0.401	0.376

TABLE III. The effects of different rescaling procedures on the losses of the model on two randomly generated datasets of the same size. The values reported correspond to the last epoch.

The first run in table (III) would seem to imply that `minmax` scaler is the optimal choice; However, this result isn't consistent throughout all runs (as can be seen in second run in table (III)) and further research is needed in determining the root cause of this effect.

CONCLUSIONS

In this short paper we ([5]) have explored and enhanced the neural network proposed during the lectures. We started by studying the performance of the original model when trained on decreasing and increasing amounts of data. As could be expected the model trained on less data was less successful in learning the data, whereas the model trained on more data was able to reach satisfactory loss values on both the training set and the validation set within the maximum number of epochs allowed. We also developed an original augmentation procedure which was successful, in fact, the model trained on the augumented dataset had and almost identical learning curve than the model trained on real-world generated data. Subsequently we probed into finding a better model structure, i.e. a neural network with different hyper-parameters, by using a grid search technique. The best model structure found was then compared to the original model on a dataset generated by another non-linear function with satisfying results. Finally the effect of different rescaling procedures was found to have no statistically noteworthy significance in the performance of the model. All the code used for this paper is available at the following `github repository`.

[1] "Learning curve models and applications: Literature review and research directions" Michel JoseAnzanello.

[2] "Deep Learning" Yoshua Bengio, Aaron Courville

[3] "A high-bias, low-variance introduction to Machine Learning for physicists" Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre G.R. Day, Clint Richardson

[4] "Random Search for Hyper-Parameter Optimization" James Bergstra, Yoshua Bengio

[5] Everyone in the group contributed equally in the intellectual discussion, further details can be provided if required.