Group 2211 – An introduction to feed-forward deep neural networks using Keras

Attar Aidin, Amjadi Bahador, Joulaei Vijouyeh Roya, and Roshana Mojtaba

(Dated: March 19, 2022)

Deep Learning is a leading framework of Machine Learning which nowadays, is commonly used in numerous applications. *Artificial Neural Networks* (*ANN*s) are inspired by human brain. Through the last years, *ANN*s have acquired a profile defined by multiple disciplines and despite this diversity, all of them preserve the biological paradigm. In this paper, we provide an overview of the methods used to train a *Deep Neural Network* (*DNN*) for a binary classification task. This review is organized into four parts: (1) Behaviour of the *DNN* concerning the training set size; (2) Tuning of the hyperparameters of the model; (3) Usage of different scaling and weight initialization methods; (4) Usage of a different data distribution for training process. The study is carried out using the `Keras` library. Results may be surprising at first glance, but surely recalls the complexity of the Deep Learning tasks.

Introduction

Neural networks (*NN*) are neural-inspired nonlinear models in the area of supervised learning. The basic unit of a neural network is a stylized as neuron i that takes a vector of d input features $x = (x_1, x_2, \dots, x_d)$ and produces a scalar output $a_i(x)$. A *NN* consists of many such neurons stacked into layers, with the output of one layer serving as the input for the next one [1]. A representation of the structure of an *ANN* is shown in Figure 1.

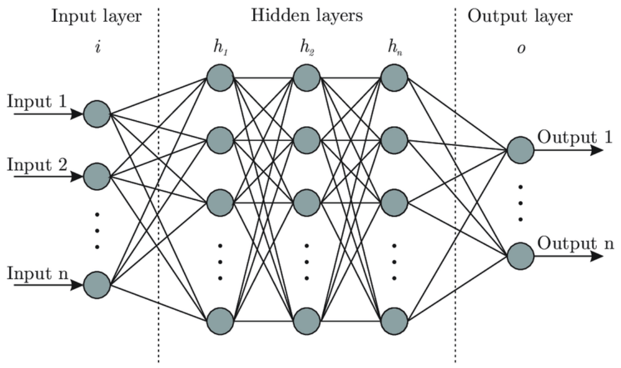

Figure 1. Basic structure of an ANN [2]

When dealing with *DNN*s, one of the most challenging problems is having a sufficient amount of data samples to be sure the network would be properly trained. In reality, it is usually difficult to collect a large number of training samples, and less number of samples increases the chance of overfitting [3].
Various techniques have been developed to solve the problem of overfitting: enlarging the dataset with artificially generated data, implementing techniques such as Transfer Learning to transfer the trained network (*CNN*) from the large sample dataset to a small sample dataset for secondary training, the use Pooling algorithms instead of fully connected networks, and many others. These methods partially solve the problem of the small-sized datasets in deep learning and improve the operation efficiency [3].

In this article, we analyze how different datasets with various dimensions would affect the training process of a *DNN*; what kind of problems we usually encounter when dealing with deep neural networks and more importantly, which techniques can be incorporated to avoid such problems.

Methods

Several artificially generated datasets are used to train our *DNN* model. In particular, two different algorithms are used for data generation. For each generating algorithm, we study the training accuracy of the *DNN* using different dimensions for the training set, including one augmented dataset. After splitting the samples of the regular set into training and validation sets, we generate the new augmented samples similar to the real ones as follows

$$x = [x_1, x_2] \longrightarrow x' = [x_1 + s_1, x_2 + s_2]$$

where $[s_1, s_2]$ are small random shifts generated according to a Normal distribution centred in 0; thus we generate for each real sample 10 generated ones.

The *DNN* models are constructed using the `Keras` library: to instantiate the model we use the `Sequential()` method and we add different deep layers one by one as follows:

```python
def create_DNN(activation, dropout_rate, layers):
    model = Sequential()
    model.add(Dense(L,input_shape=(L,),activation=activation))
    for i in range(len(layers)):
        model.add(Dense(layers[i],activation=activation))
    model.add(Dropout(dropout_rate))
    model.add(Dense(1,activation="sigmoid"))
    return model
```

We use the `add()` method to attach layers to our model. Every `Dense()` layer accepts as its first required argument an integer that specifies the number of neurons. The type of activation function for the layer is defined using the activation optional argument. Here We use the `relu`, and the `sigmoid` functions.

In order to make the DNN work properly, we have to make sure that the numbers of input and output neurons for each layer match. Therefore, we specify the shape of the input in the first layer of the model explicitly. The sequential construction of the model then allows Keras to infer the correct input/output dimensions of all hidden layers automatically [1].

After the instantiation of the model, we specify the loss functions and required metrics, using the function compile(). The loss function should be determined based on the kind of hypothesis class we are dealing with. For a classification problem, the best candidates are usually cross-entropy and logarithmic loss functions. These loss functions measure the performance of a classification task not in a discrete 0-1 way but as a continuous value between 0 and 1. Cross-entropy loss increases as the predicted probability diverge from the true label [4], and, according to [1], is given by

$$C = -\sum_{i=1}^{n} [y_i \log \hat{y}_i[\mathbf{w}] - (1-i)\log(1-\hat{y}_i[\mathbf{w}])] \tag{1}$$

with $\hat{y}_i[\mathbf{w}] = p(y_i = 1|\mathbf{x}_i;\mathbf{w})$ the probability that data point i is predicted to be in category 1. Since we deal with a binary cast, we choose the binary_crossentropy defined in Keras' losses module.

To optimize the weights of the network, we choose the Adam optimizer implemented in Keras, according to [5]. To test the performance of the model, one can look at the accuracy metric, defined, according to [5], as the percentage of correctly classified data points.

```
def compile_model(optimizer=tf.keras.optimizers.Adam(),
                  activation="relu",dropout_rate=0.2,layers=(20,20)):
    model=create_DNN(activation,dropout_rate,layers)
    model.compile(loss=keras.losses.binary_crossentropy,
                  optimizer=optimizer,
                  metrics=['accuracy'])
    return model
```

The training procedure of the *DNN* is performed using SGD with mini-batches, adding a layer of stochasticity, computing the gradient on random groups with smaller size than the total number of data points.

Shuffling the dataset during the training process improves the stability of the model. Thus, we do it over a number of training epochs.
We perform the training process using the function fit() from the Keras library.

To have an estimation of the generalization error of our model, We use validation_data. At the end of each epoch, we evaluate the value of loss and any model metrics. Note the fact that the validation loss of data provided using validation_data is not affected by regularization layers like noise and dropout [5].

```
fit=model.fit(x_train, y_train,
              epochs=nepoch, batch_size=50,
              validation_data=(x_valid,y_valid),
              verbose=1)
```

To calibrate the hyper-parameters used in our model to optimize the performance of the model, we use the GridSearchCV() function of scikit-learn. In particular, to avoid a drastically long computational time, we divide this process into two parts: the former to find the best optimizer and activation function, and the latter to find the best architecture in terms of the number of neurons for each layer and dropout rate.

```
model_gridsearch=KerasClassifier(build_fn=compile_model)
optimizer=['SGD', 'RMSprop', 'Adagrad', 'Adadelta',
                     'Adam',  'Adamax',    'Nadam']
activation=['softmax', 'softplus',     'softsign',  'relu',
            'tanh',  'sigmoid', 'hard_sigmoid', 'linear']
param_grid = dict(optimizer=optimizer, activation=activation)
grid=GridSearchCV(estimator=model_gridsearch,
                  param_grid=param_grid,n_jobs=1,cv=4)
grid_result=grid.fit(x_train,y_train,epochs=100,
                     batch_size=50,shuffle=True,verbose=0)
```

```
model_gridsearch=KerasClassifier(build_fn=compile_model)
dropout_rate = [.0, .1, .2, .3, .4,
                .5, .6, .7, .8, .9]
layers       = [   (20,20), (30,30),    (80,80),
                (20,30,20), (40,20), (10,20,30)]
param_grid=dict(dropout_rate=dropout_rate,layers=layers)
grid=GridSearchCV(estimator=model_gridsearch,
                  param_grid=param_grid,n_jobs=1,cv=4)
grid_result=grid.fit(x_train,y_train,epochs=100,
                     batch_size=50,shuffle=True,verbose=0)
```

The code was developed in Python 3.7 using a Jupyter framework, and it's available here.

Results

In this last section, we discuss a collection of the results obtained so far. For easier illustration, this part is divided into 4 distinct subsections.

Part (I): Response to the training set size

In the first part, we examine the performance of the neural network when fed with different training set sizes.

An initial visualization of the dataset used is reported in Figure 2(a). We use several dimensions for the datasets to study the training efficiency of the *DNN* as a function of the number of samples used. The number of samples for each set is reported in Table I. Figure 2(c) shows the training curves for each dataset size.

Table I. Different sizes of the sets used to train the network, split between the training set and the validation set.

Set	Training set	Validation set
Super-Reduced	400	100
Reduced	1600	400
Regular	3200	800
Increased	6400	1600
Super-Increased	12800	3200
Augmented	32000	800

We observe that the goodness of the training procedure is very sensitive to the initial weights, resulting in episodes in which better results are obtained using smaller datasets. For each set, we study this phenomenon by repeating the model fitting for each dataset 100 times each to study the average behaviour of the *DNN*. The results for the training and validation accuracies are reported in the *BoxPlots* in Figure 2(d), while the plots for

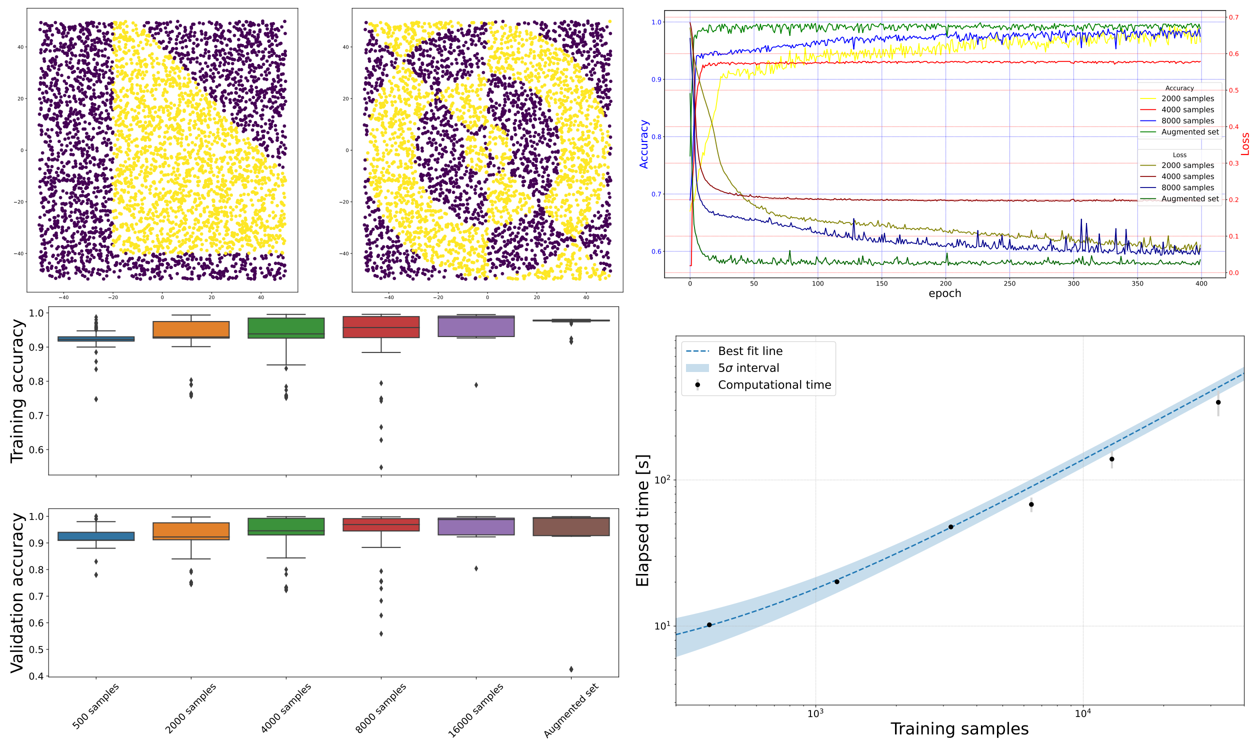

Figure 2. (a) Top left: Dataset generated using the first non-linear function; (b) Top center: Dataset generated using the second non-linear function; (c) Top right: training curves for the validation accuracy and loss for the reduced, regular, increased and augmented datasets. We do not report the super-reduced and the super-increased due to lack of space. (d) Bottom left: Training and validation accuracy for each size of the dataset. While the average value is monotonically increasing, the dispersion of it does not follow a decreasing pattern. There are several outliers for each dataset; in particular the *augmented* set suffers of several cases of overfitting over the training set, which have catastrophic results on the validation set. (e) Bottom right: Calculation time for different size of datasets and the corresponding linear fit [note the logscale].

the losses are not reported due to the lack of space. The fitting repetition was possible with the help of parallel-running of the procedure on several computers. We observe that, even if the average accuracy grows with the number of samples, larger sets correspond also to more dispersed results, other than some overfitting problems.

Computational time

We do a further study over the computational time as a function of the number of samples used to train the *DNN*. For each set size, we train the same network (2 hidden layers of 20 neurons, `relu` activation, `Adam` optimizer) 7 times and we pair it with the average and the error of the mean of the time obtained. Figure 2(e) shows the results of the study and the corresponding linear fit, which is a compatible hypothesis.

Part(II): Tuning of the hyperparamteres

The choice of the hyperparameters has to be performed carefully. Indeed, while for the simple distribution we use, the results are satisfying even for a small network, for more complex ones (as we discuss in Part(IV)), without a proper choice of the parameters the results are catastrophic.

For this task we use `GridSearchCV`, tuning several parameters, as can be seen in Table III.

A visual comparison between the fit results using the best parameters and the initial ones is reported in Figure 3(a).
Anyway as we previously explained, the training procedure is very sensitive to the initial weights and it can happen that the best architecture found gives worse results than some others. We should do an average grid-search, taking the score for each parameter combination for several times. In the end we evaluate the average best. Unfortunately, the computational time of the grid search makes it impossible for us performing more than 2/3 runs which is not a sufficient number to build a solid statistic.

Part(III): Rescaling and Initialization

Scaling the dataset is an essential pre-processing step when working with *DNN*s. Also, the initial weights of the network can be determined by using different statistical distributions. We incorporate different scaling methods and weight initializing distributions which are listed in Table II.

To achieve stability in results we take average of 40 times of running each configuration. The results of us-

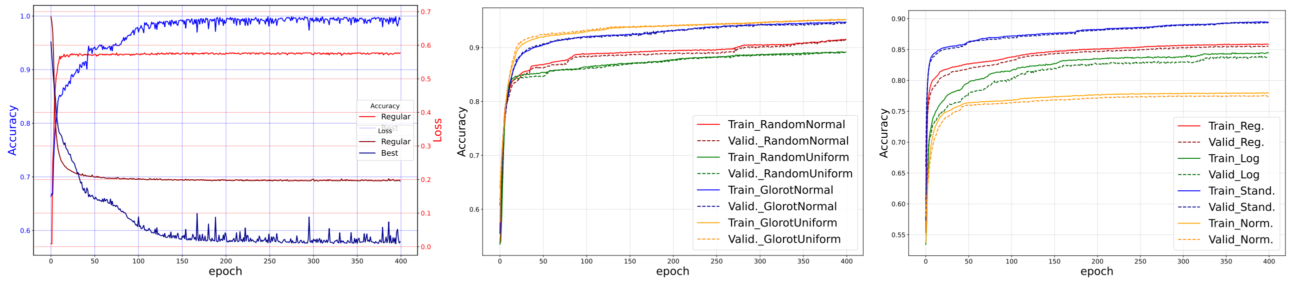

Figure 3. (a) Left: Accuracy and loss for regular parameters and best parameters found out using the Grid-Search algorithm; (b) Center: Accuracy for different initializing weights methods; (c) Right: Accuracy for different rescaling methods

ing various kernel initializers and scaling functions are shown in Figures 3(b) and 3(c) respectively. In the case of initializers, GlorotNormal gains the highest value for accuracy after 400 epochs. And in terms of scaling methods, Standardization leads to the best accuracy.

Table II. Different scaling & weight initialization methods, with $L = \sqrt{6/(\alpha+\beta)}$, and α,β, number of input and output units in the weight tensor

Scaling	$F(x)$	In. weights	$F(w)$
Reg.	$x/50$	Rand.Normal	$\mathcal{N}(\mu,\,\sigma^2)$
Log	$\log x$	Rand.Uniform	$\mathcal{U}(a,\,b)$
Norm.	$\frac{x-x_{min}}{x_{max}-x_{min}}$	Glor.Normal	$\mathcal{N}\left(0,\sqrt{\frac{2}{\alpha+\beta}}\right)$
Stand.	$\frac{x-\overline{x}}{\sigma}$	Glor.Uniform	$\mathcal{U}(-L,+L)$

Part(IV): Usage of a different data distribution

We repeat the study done for the first distribution for a new one. A visualization of the new dataset is shown in Figure 2(b). As can be seen, the new distribution is more complex and the study requires the use of a larger dataset (8000 samples are used) and indeed, a more complex network architecture. To calibrate the parameters, we use the grid-search process, in a similar way as explained in the previous sections. The best parameters found are reported in Table III

Table III. Best parameters found using `gridsearch` for the first and the second data distributions

Parameter	I distrib.	II distrib.
`optimizer`	`Adam`	`Adam`
`activation`	`relu`	`relu`
`layers`	`(80,80)`	`(10,20,30,40,30,20)`
`dropout_rate`	`0`	`0`

The training curves are reported in Figure 4. As can be, seen we use 2000 epochs, more than the other analysis made, which is due to larger complexity of the new distribution used.

Conclusions

The implementation of neural networks, as shown by the results, is generally capable of achieving high degrees of accuracy when dealing with simple binary classification tasks. However, the training process of *DNN*s is in no way straightforward, but a challenging task.

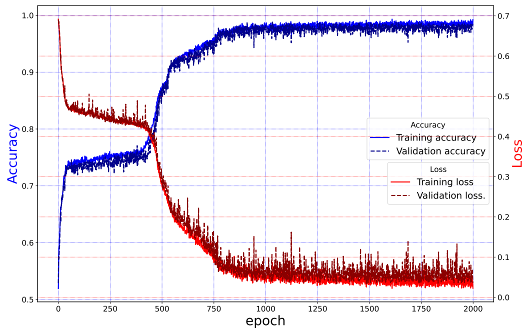

Figure 4. Training curves for the training and validation accuracy and loss for the second non-linear function used

Including different size training sets, we have shown that increasing the number of samples does not always lead to significantly better accuracies, but unnecessarily adds to the computational time. Anyhow, the use of the augmentation process to artificially increase the dimensions of the dataset generally leads to better results. Anyway, precautions should be taken to avoid cases of overfitting.

Additionally, we remark the importance of the grid-search processing as a proper tool for tuning the hyper-parameters of the model; especially when dealing with complex problems. On the other hand, concerning simpler tasks, the way we initialize the weights of the model seems to be more relevant. Moreover, we incorporate different data scaling methods. It turns out that scaling our dataset using a proper function is relatively a costless act that can be done to further improve the accuracy of our model.

[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. Bre, J. Gimenez, and V. Fachinotti, Energy and Buildings **158** (2017), 10.1016/j.enbuild.2017.11.045.

[3] W. Zhao, AIP Conference Proceedings **1864**, 020018 (2017), https://aip.scitation.org/doi/pdf/10.1063/1.4992835.

[4] M. L. Glossary, "Loss functions," (2015).

[5] F. Chollet *et al.*, "Keras," (2015).