Deep Learning

Practical Project : Building a deep neural network

Mehdi Gati
Smaïl Darbane
Franko Sikic

April 15, 2020

Contents

1 Introduction

Deep learning is part of a broader family of machine learning methods based on artificial neural networks with representation learning. Deep learning architectures such as deep neural networks, reccurent neural networks and convolutional neural networks have been applied to several different fields including computer vision, speech recognition, natural language processing, etc. A deep neural network is an artificial neural network with multiple layers between the input and output layers.

This assignment consists of building a deep artificial neural network to estimate the approximate location of housing blocks. The objective of the activity is to follow the construction process of a deep artificial neural network for a classification problem using the California Housing Prices dataset. This is the dataset used in the second chapter of Aurélien Géron's recent book 'Hands-On Machine learning with Scikit-Learn and TensorFlow'. It serves as an excellent introduction to implementing machine learning algorithms because it requires rudimentary data cleaning, has an easily understandable list of variables and sits at an optimal size between being to toyish and too cumbersome.

The dataset contains information from the 1990 California census. It consists of 20641 rows (with header) and 10 columns labeled as: longitude, latitude, housingMedianAge, totalRooms, totalBedrooms, population, households, medianIncome, medianHouseValue and oceanProximity. As already mentioned, this classification problem consists of estimating the approximate location of housing blocks. The approximate location is represented with a discrete variable oceanProximity which may have one of five possible values: NEAR BAY, ¡1H OCEAN, INLAND, NEAR OCEAN and ISLAND.

2 Data Preprocessing

The following actions were performed in order to clean and prepare the dataset:

- Remove the instances corresponding to the value oceanProximity=ISLAND because there is not enough data for the learning process
- Remove instances with missing values
- Randomize the order of appearance of rows
- Max-min scale the data
- Encode the oceanProximity values according to a classification task

After the cleaning process, there were 20428 rows left (with the header).

3 Design Process

3.1 Hyperparameters

Model Hyperparameters are instead properties that govern the entire training process. They include variables which determines the network structure, and the variables which determine how the network is trained (for example, Learning Rate). Model hyperparameters are set before training (before optimizing the weights and bias).
In the globality of our neural network models we decided to opt for similar hyperparameters, that we will detail except for the number of epochs that we modified for the final model in order to refine the results.

Learning rate

If the learning rate is low, then training is more reliable, but optimization will take a lot of time because steps towards the minimum of the loss function are tiny. If the learning rate is high, then training may not converge or even diverge. Weight changes can be so big that the optimizer overshoots the minimum and makes the loss worse. We decided to use a learning rate of 0.1 .

Number of layers

After reading a few scientific articles and forums on the internet, a few rules came out in order to have an approximate idea about the number of layers to be used:
In our case we used 6 hidden layers and each layer contain respectively 2000,1000,500,250,75 and 25.

Number of epoch and Batch Size

At the beginning we decided to keep the number of epochs as being the same as the example we did in class but we felt that it was not enough to create a deep neural network, so we increased the number to 1000 and then to 2000, for the Batch Size we decided to keep 500.

Table: Determining the Number of Hidden Layers

Num Hidden Layers	Result
none	Only capable of representing linear separable functions or decisions.
1	Can approximate any function that contains a continuous mapping from one finite space to another.
2	Can represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy.
>2	Additional layers can learn complex representations (sort of automatic feature engineering) for layer layers.

Figure 1: Determining the number of hidden layers

Activation functions

In a neural network, the activation function is responsible for transforming the summed weighted input from the node into the activation of the node or output for that input.
The linear activation function overcomes the vanishing gradient problem, allowing models to learn faster and perform better.Thats why we decided to use ReLU major benefits of ReLUs are sparsity and a reduced likelihood of vanishing gradient. But first recall the definition of a ReLU is h=max(0,a) where a=Wx+b.
One major benefit is the reduced likelihood of the gradient to vanish. This arises when a¿0. In this regime the gradient has a constant value. In contrast, the gradient of sigmoids becomes increasingly small as the absolute value of x increases. The constant gradient of ReLUs results in faster learning.
The other benefit of ReLUs is sparsity. Sparsity arises when a0. The more such units that exist in a layer the more sparse the resulting representation. Sigmoids on the other hand are always likely to generate some non-zero value resulting in dense representations. Sparse representations seem to be more beneficial than dense representations.

3.2 Guidelines to choose hyperparameters

Bias-Variance Tradeoff is one of the important aspects of Applied Machine Learning. It has simple and powerful implications around the model complexity and its performance.

We say there's a bias in a model when the algorithm is not flexible enough to generalize well from the data. Linear parametric algorithms with low complexity such as Regression and Naive Bayes tend to have a high bias.

Variance occurs in the model when the algorithm is sensitive and highly flexible to the training

data. Non-Linear non-parametric algorithms with high complexity such as Decision trees, Neural Network, etc tend to have high variance.

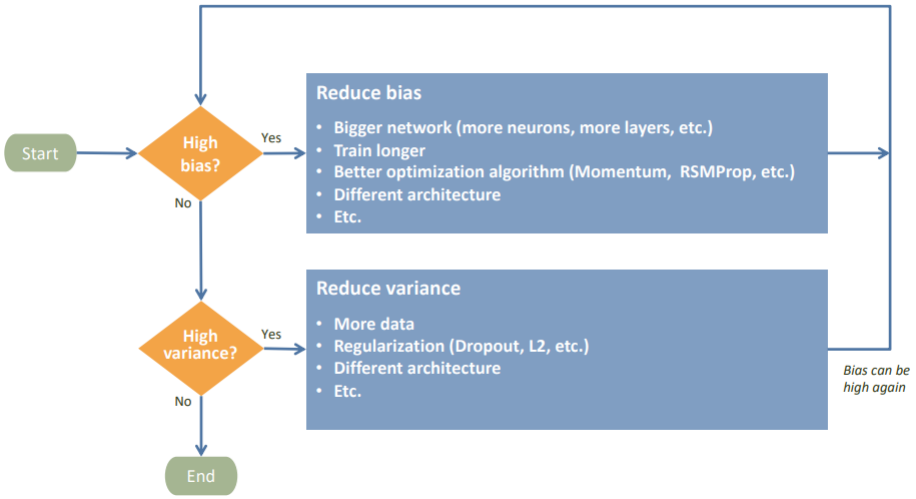

Figure 2: The different steps to improve the accuracy of the model

This graph represents the different steps to be taken to better manage the trade-off between bias and variance. We will use this graph as a reference during the construction of our prediction model.

4 Model Progression

4.1 First Model

In our first model the architecture was not classical, we opted for an Adamax optimizer, The idea with Adamax is to look at the value v as the L2 norm of the individual current and past gradients. We also added the dropout which is regularization technique to avoid overfitting thus increasing the generalizing power. We then used the Batch Normalization function which is also used to avoid overfitting problems, we can see the model and the results below.

Model & Results

As we can see in the figures below, the results are not satisfactory, the adamax optimizer was not the right one to use, we also made the mistake of using the Dropout before the Normalization batch.

```
n_neurons_per_hlayer = [2000,1000, 500, 250, 75, 25]
learning_rate = 0.1
```

```
model = keras.Sequential(name="my_model")
model.add(keras.layers.InputLayer(input_shape=(INPUTS,)))
for neurons in n_neurons_per_hlayer:
  model.add(keras.layers.Dense(neurons, activation="relu"))
  model.add(keras.layers.Dropout(rate=0.2))
  model.add(keras.layers.BatchNormalization())
opt = keras.optimizers.Adamax(lr=0.4, beta_1=0.9, beta_2=0.9999)
```

Figure 3: Model

```
Error (training):  58.0 %
Error (development test):  56.41 %
Time:  234 seconds
```

Figure 4: Results

4.2 Second Model

In the second model we only thought about changing the optimizer in order to understand where our error really was and the results were obvious, we opted for the SGD optimizer of nesterov, which gave better results but we still didn't realize that we had made a mistake by putting the dropout first before the Batch Normalization.

Model & Results

```
[5]  n_neurons_per_hlayer = [2000,1000, 500, 250, 75, 25]
     learning_rate = 0.1
```

```
[6]  model = keras.Sequential(name="my_model")
     model.add(keras.layers.InputLayer(input_shape=(INPUTS,)))
     for neurons in n_neurons_per_hlayer:
       model.add(keras.layers.Dense(neurons, activation="relu"))
       model.add(keras.layers.Dropout(rate=0.2))
       model.add(keras.layers.BatchNormalization())
     model.add(keras.layers.Dense(OUTPUTS, activation="softmax"))
```

Figure 5: Model

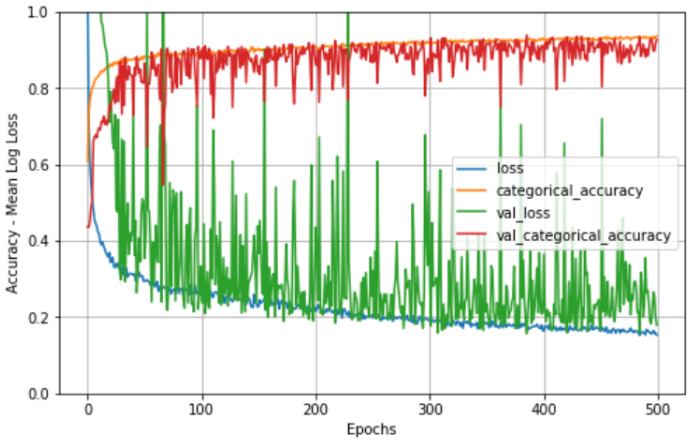

Figure 6: Graph results

```
Error (training):  6.32 %
Error (development test):  7.24 %
Time:  208 seconds
```

Figure 7: Graph results

4.3 Third Model

In this one we used Dropout function after the Batch Normalization, an experimental test seems to suggest that ordering does matter. We ran the same network twice with only the batch norm and dropout reverse. When the dropout is before the batch norm, validation loss seems to be going up as training loss is going down. They're both going down in the other case.

Model & Results

```
[5]  n_neurons_per_hlayer = [2000,1000, 500, 250, 75, 25]
     learning_rate = 0.1
```

```
model = keras.Sequential(name="my_model")
model.add(keras.layers.InputLayer(input_shape=(INPUTS,)))
for neurons in n_neurons_per_hlayer:
  model.add(keras.layers.Dense(neurons, activation="relu"))
  model.add(keras.layers.BatchNormalization())
  model.add(keras.layers.Dropout(rate=0.2))
model.add(keras.layers.Dense(OUTPUTS, activation="softmax"))
opt = keras.optimizers.SGD(lr=0.005, decay=1e-6, momentum=0.9, nesterov=True)
model.summary()
```

Figure 8: Model

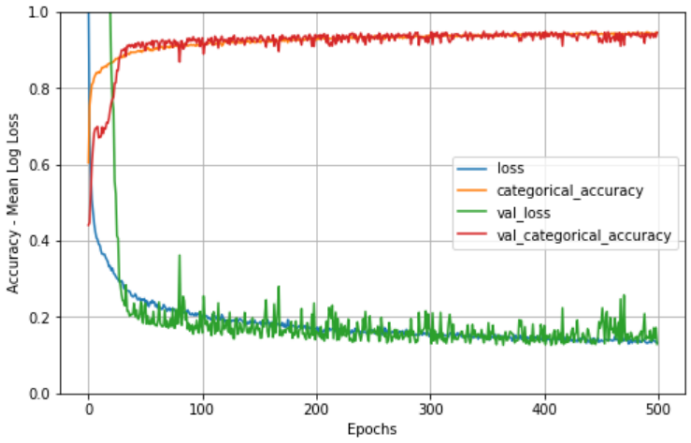

Figure 9: Graph results

```
Error (training):  5.4 %
Error (development test):  5.36 %
Time:  232 seconds
```

Figure 10: Graph results

4.4 Fourth Model

In this model we tried to add a kernel initializer.The aim of weight initialization is to prevent layer activation outputs from exploding or vanishing during the course of a forward pass through a deep neural network. If either occurs, loss gradients will either be too large or too small to flow backwards beneficially, and the network will take longer to converge, if it is even able to do so at all.

Model & Results

```
[44] n_neurons_per_hlayer = [2000,1000, 500, 250, 75, 25]
     learning_rate = 0.1
```

```
[87] model = keras.Sequential(name="my_model")
     model.add(keras.layers.InputLayer(input_shape=(INPUTS,)))
     for neurons in n_neurons_per_hlayer:
       model.add(keras.layers.Dense(neurons, activation="relu",kernel_regularizer=keras.regularizers.l2(0.001)))
       model.add(keras.layers.BatchNormalization())
       model.add(keras.layers.Dropout(rate=0.2))
     model.add(keras.layers.Dense(OUTPUTS, activation="softmax"))
     opt = keras.optimizers.SGD(lr=0.005, decay=1e-6, momentum=0.9, nesterov=True)
     model.summary()
```

Figure 11: Model

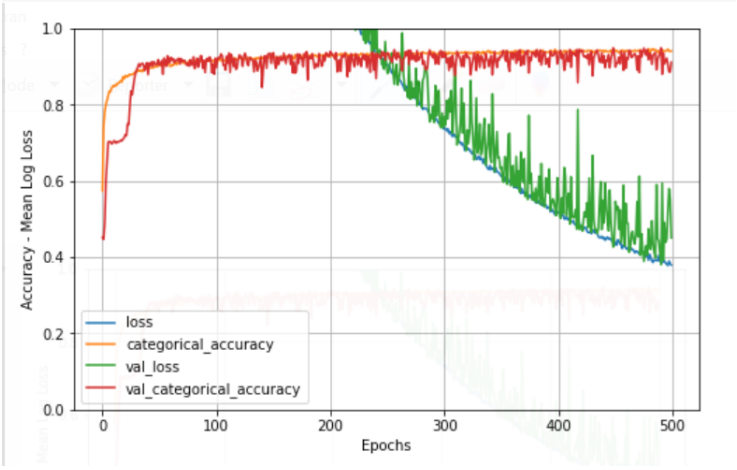

Figure 12: Graph results

```
Error (training):  5.92 %
Error (development test):  8.88 %
Time:  252 seconds
```

Figure 13: Graph results

Discussion

It is apparent that the results are not good even using L2 standardization.

5 Final Results

In our final model, we were able to make all the modifications we noticed on the previous models, namely the optimizer, the order of the functions, the activation function. We trained our neural network on 4000 epoch.

Model & Results

```
n_neurons_per_hlayer = [2000,1000, 500, 250, 75, 25]
learning_rate = 0.1
```

```
model = keras.Sequential(name="my_model")
model.add(keras.layers.InputLayer(input_shape=(INPUTS,)))
for neurons in n_neurons_per_hlayer:
  model.add(keras.layers.Dense(neurons, activation="relu"))
  model.add(keras.layers.BatchNormalization())
  model.add(keras.layers.Dropout(rate=0.2))
model.add(keras.layers.Dense(OUTPUTS, activation="softmax"))
opt = keras.optimizers.SGD(lr=0.005, decay=1e-6, momentum=0.9, nesterov=True)
model.summary()
```

Figure 14: Model

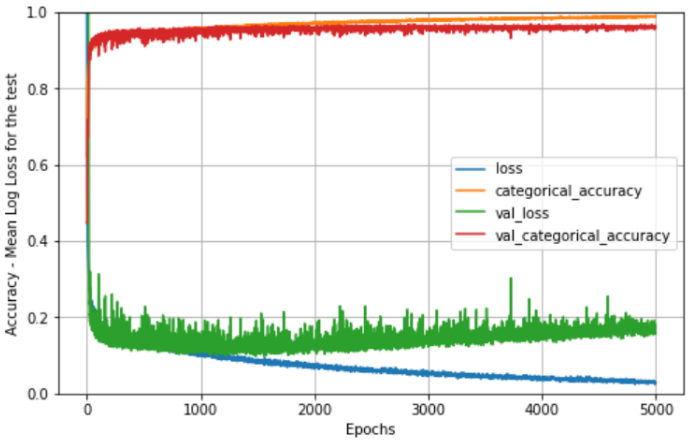

Figure 15: Graph results for training and development data

```
Error (training):  1.09 %
Error (development test):  3.72 %
Time:  3344 seconds
```

Figure 16: Graph results for training and development data

```
2043/2043 [==============================] - 0s 81us/sample - loss: 0.1629 - categorical_accuracy: 0.9618
CNN took 0.1728522777557373 seconds
Test loss: 0.1629182938308552 - Accuracy: 0.96182084
```

Figure 17: Final test accuracy and loss

.

Accuracy	0,9618
E_train	0,0109
E_test	0,0372
E_human	0,0382
Bias	0,0273
Variance	0,0263

Figure 18: Bias & Variance of the final model

.

Discussion

The results obtained are very successful. As can be seen in figure 18, we have the two values of bias and variance that are very low.

6 Conclusion

In short, we have been able to model our neural network according to the previous models and what we have been able to study in the first chapter where we understood the importance of deep neural networks but also the importance that an activation function can play in saturation, we also became aware of the usefulness of hyperparameters then in the second chapter we knew how to optimize and improve our neural network in order to achieve more than satisfactory results.