Fundamentals of Data Science Final Project Full Report

SAPIENZA
UNIVERSITÀ DI ROMA

Project Title:

Dog Breed Identification

The Supervisor:

Prof. Fabio Galasso

Developers:

Mehrzad Jafari Ranjbar

Marco Muscas

Giovanni Giunta

Francesco Lauro

Shokoufeh Mansourihafshejani

Academic Year: 2020/2021

Table of Contents

Overview

Dog Breed Identification is a competition on Kaggle website aimed at creating and training a model to determine the breed of a dog in an image. For this competition, the website provides a folder of 10,000+ images as training set and their labels as a separate .csv file, plus another folder with test images without any labels.

This is clearly an image classification problem, with 120 unique dog breeds (i.e. 120 classes) which we have tried to solve it using TensorFlow and Google Colaboratory.

To start our project, we first need to preprocess all the images we have in our training dataset. The idea behind preprocessing is to retrieve the pictures we have in our training dataset from Google Drive and store them in the form of Tensors. In other words, we convert the data contained in the image files into numbers and store them into Tensor variables. This way we can leverage the power of GPUs parallel computation, like the ones offered by Google Colaboratory's virtual machines, which we will make use of in order to take advantage of a much faster computation. We could also use the Kaggle's API, but we have used Google Drive to present a more comprehensive approach, since not all the projects have APIs.

A second step is to resize our images by using the img_load module from Keras, because the chosen models only works with specific image sizes. In our assignment, we decided to explore two different approaches:

- In the first one, we chose to follow the Linear approach. To do so, we selected the **mobilenet_v2_130_224** model from TensorFlow Hub that comes from the **MobileNet** architecture, that accepts (224,224,3) as an eligible input size.
- In the second one, we used 4 layers, respectively **InceptionV3**, **Xception**, **NASNetLarge**, **InceptionResNetV2**, to achieve a Deep Neural Network (DNN) approach. Image size (331,331,3)

A clear result we obtained from implementing both strategies, is that the DNN approach returns much more accurate results. In fact, it is hard for the Linear approach to correctly predict 120 different classes (we only got as validation accuracy a value of 0.67). We have tried to fine-tune our model and also changed some parameters (e.g. learning rate). Despite our efforts, we couldn't reach a higher accuracy level, therefore, we decided that we needed a much more advanced solution and kept experimenting.

To solve this poor accuracy level, we started building our own DNN, made of 4 different layers, to extract and collect the features of the images, and, using the retrieved features to make predictions over the entire dataset. With this approach we achieved 0.94 as validation accuracy, hence improving our model significantly.

In our submission we have provided **two Jupiter notebooks**, each one containing one of these two different approaches. When submitting our predictions on Kaggle, we got a score of 0.88653 with the Linear approach (first notebook), and 0.18703 with the DNN approach (second notebook).

In the following pages, we'll explain in detail the modules and techniques that we have used for these both approaches.

Importing and processing input data

Importing and unzipping data

There are a few ways we could import and unzip the file, since the data we're using is hosted on Kaggle, we could even use the Kaggle API which is great but what if the data you want to use wasn't on Kaggle?

The method we used is to mount our drive in the notebook and import the zip file.

We have seen that in the zip file we have the following items:

- A folder named “train” with 10.2k images of different dog breeds
- A folder named “test” with 10.4k images of different dog breeds
- A labels.csv file containing all the filenames in train folder and their corresponding breeds
- A sample_submission.csv file showing the sample data frame for submission on Kaggle website.

Preprocessing images and building one-hot array labels

After unzipping the files, we have selected the unique dog breeds (120 classes) and using all the data from “labels.csv” and to_categorical module from TensorFlow Keras utilities (tf.keras.utils), we have created our one hot encoding arrays for training set to be our target variable (y) for the train and evaluation of our model. We will use 10% of them as validation set to evaluate our model.

More in detail, we associated to each unique dog breed its unique label and proceeded with creating one hot arrays for each image in the dataset. In this case, each one hot array has length equal to the number of unique dogs’ breeds and has value “1” at the index corresponding to the position of its dog breed in the unique dog breeds vector, and “0” otherwise.

For the preprocessing of images, we have created a custom function called “preprocess_train” that takes the image path, their labels and the desired size of images as input and perform the following steps:

1- Read image directory.
2- Resize it, then stack it into one big NumPy array. Set to (224,224,3).
3- Read sample's label form the labels data frame.
4- One hot encodes labels array.
5- Shuffle Data and label array.

In the function above, we have used load_img from tensorflow.keras.preprocessing.image to use the image directory, setting the desired image size as input to load our images into PIL format. Therefore, we’ll have our independent variable (X) which is our images transformed into tensors, and target variable (y) that is the boolean one-hot array representation of each dog breed corresponded to training images.

Until now, all the steps were the same in both notebooks. In the following pages we’ll explain our approach and review how we have chosen those approaches.

The first approach: Using MobileNet V2

Now our data is ready, let's prepare it to model. We'll use an existing model from TensorFlow Hub. TensorFlow Hub is a resource where you can find pretrained machine learning models for the problem you're working on. Using a pretrained machine learning model is often referred to as **transfer learning**.

Why did we use a pretrained model?

- Building a machine learning model and training it on lots from scratch can be expensive and time consuming.

Transfer learning helps eliminate some of these by taking what another model has learned and using that information with your own problem.

How do we choose a model?

- Since we know our problem is image classification (classifying different dog breeds), we can navigate the TensorFlow Hub page by our problem domain (image).

We start by choosing the image problem domain, and then can filter it down by subdomains, in our case, image classification. Doing this gives a list of different pretrained models we can apply to our task. Clicking on one gives us information about the model as well as instructions for using it.

For example, clicking on the **mobilenet_v2_130_224** model that we have used for this section, tells us this model takes an input of images in the shape 224, 224. It also says the model has been trained in the domain of image classification.

How to build the model? (Sequential Approach)

We have created a custom function called “create_model” to set our layers and compile the model.

First, we have used Sequential module from TensorFlow Keras library to set the following:

1. Passing the URL of our model to hub.KerasLayer to instruct our pre-train model
2. Using tf.keras.layers.Dense by assigning the “number of our classes” (i.e. 120) to “unit” and “softmax” as our “activation”

Why did we use softmax as our activation function?

- “softmax” or “softargmax” is a generalization of the logistic function for multiple dimensions or classes. By using it, our predictions could be interpreted as arrays of probability distributions. Therefore, for each prediction we are able to review which probability was given to each breed.

Second, we have used “.compile” as a method of our model to introduce the following variables for compiling and setting the desired configuration of the model for training. As our arguments we have used the following:

1. Setting the loss as Categorical Cross Entropy using tf.keras.losses.CategoricalCrossentropy() because we have many classes to predict, and our labels are given as a one hot representation.
2. We used “Adam” as our “optimizer” using tf.keras.optimizers.Adam(), which is a stochastic gradient descent method which we used its default learning value, that is 0.001.
3. As accuracy metric we used the accuracy class, that calculates how often predictions equal true labels as a binary representation, and to make sure that our model is actually learning.

After creating our model, we have checked the summary of the model using “model.summary()”:

```
Building model with: https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/4
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
keras_layer (KerasLayer)     (None, 1001)              5432713
_________________________________________________________________
dense (Dense)                (None, 120)               120240
=================================================================
Total params: 5,552,953
Trainable params: 120,240
Non-trainable params: 5,432,713
```

The non-trainable parameters are the patterns learned by mobilenet_v2_130_224 and the trainable parameters are the ones in the dense layer we added. This means the main bulk of the information in our model has already been learned and we're going to take that and adapt it to our own problem.

Our next step was to set our callbacks. Callbacks are useful functions a model can use during training to do things such as:

- Save a model’s progress
- Check a model’s progress
- Stop training early if a model stops improving.

The two callbacks we're going to add are a “TensorBoard” callback and an “EarlyStopping” callback.

TensorBoard helps provide a visual way to monitor the progress of the model during and after training. It can be used directly in a notebook to track the performance measures of a model such as loss and accuracy.

To set up a TensorBoard callback and view it in a notebook, we had to do three things:

1. Load the TensorBoard notebook extension. Using **%load_ext tensorboard** magic function
2. Create a TensorBoard callback which is able to save logs to a directory and pass it to our model
3. Visualize our models training and validation logs, using our custom function “create_tensorboard_callback”.

For the Early Stopping callback we have used EarlyStopping module from tf.keras.callbacks that is specialized for this purpose. We have assigned the following parameters:

1. "val_accuracy" to “monitor”, that is going to be our chosen metric.

2. Setting number of “steps” to 3.

By using this method, the training loop will check at the end of every epoch whether the accuracy is no longer increasing. As we have set the steps equal to 3, meaning that if the validation accuracy doesn’t increase after 3 consecutive epochs, the training will stop.

Early stopping helps prevent overfitting by stopping a model when a certain evaluation metric stops improving. If a model trains for too long, it can do so well at finding patterns in a certain dataset that it's not able to use those patterns on another dataset it hasn't seen before (i.e. overfitting).

How to train the model?

Additional to creating the model and setting our callback functions, we had to “fit” our model with the following arguments:

1. Setting our features and target for training set
2. Setting the number of epochs to 100
3. Assigning our validation set (10% of total training data)
4. Setting our batch size to 32
5. Introducing our 2 callback functions (i.e. TensorBoard and EarlyStopping)

Results

We have seen that model started learning as for 1st epoch ended with 0.295 validation accuracy, on the 6th epoch our model became completely overfitted on training set, and accuracy value became 1.0 and on the 12th epochs, our callbacks ended the training session. Reaching 0.655 as our validation accuracy, the results were indeed poor and not accepted. We have tried by setting our learning rate to a lower number, thinking that might improve our results, even though the accuracy did not change much (i.e. 0.67). We have studied more about the problem and found out that because we have 120 different classes, having only one model is not enough to predict all of these classes correctly. Therefore, we have needed a much more complex model.

More about mobilenet_v2_130_224

mobilenet_v2_130_224 is a pre-trained model with MobileNet V2 architecture published by Google. MobileNet V2 is a family of neural network architectures for efficient on-device image classification and related tasks. This TensorFlow (TF) Hub model uses the TF-Slim implementation of mobilenet_v2 with a depth multiplier of 1.3 and an input size of 224x224 pixels.

The Second approach: Using Deep Neural Networks (DNN)

While we were searching for another solution for our model on internet and reading different documentations and checking the resources, we have found out about this notion.

A deep neural network (DNN) is an artificial neural network (ANN) with multiple layers between the input and output. Each mathematical manipulation as such is considered a layer, and complex DNN have **many layers**, hence the name "deep" networks.

Using DNNs is different from what we did in the first approach, since the former is the **Functional** approach and the latter (i.e. using single linear model) considered as **Sequential** approach, although, we have already preprocessed all our training images and also encoded our target variable as one hot arrays, even though, we had to make some adjustment in preprocessing to fasten the process.

Our method for this specific DNN method works as following:

1. Creating a feature extractor
2. Extracting features using different models (multiple layers)
3. Concatenating our features together
4. Fitting our model with the extracted features

Getting features from the images

We have created a custom function called “get_features” that takes the following arguments as its inputs:

- The model
- The data preprocessor respected to the chosen model
- The shape of our images that has been set to [331,331,3] for 331 as height and width and 3 color channels (RGB). Because that is the largest default shape that we need for our models.
- The training data that is our preprocessed images (X)

and returns the features that has been extracted from the model.

How to build the extractor (Functional Approach)

First, we had to write “get_features” function in order to achieve our new independent data. The steps of the pipeline that we have implemented is as follows:

1. We have passed our image shape to Input from tf.keras.layers and assigned it to “input_layer”
2. We have used Lambda from tf.keras.layers to wrap the respected “data preprocessor” that is the input of our function and “input_layer” that we have created in the 1st step and have assigned it “preprocessor” variable to be used in the next step. Lambda layers are best suited for simple

operations or quick experimentation, although They should only be loaded in the same environment where they were saved since they are fundamentally non-portable.

3. Passing 'imagenet' to **weights**, False to **include_top**, “input_size” to **input_shape** in our input model (model_name) and also setting the “preprocessor” to add the defined methods. We have assigned the aforementioned to a variable called “base_model” which is our initial model.
4. We have used GlobalAveragePooling2D class from tf.keras.layers that is generally used for spatial data and will return a 2D tensor with shape (batch_size, channels) that we wanted to set as our desired output and set the value to the variable “avg” to use it as our output.
5. We have used Model class from tf.keras to build our model (Functional) which we set “input_layer” as **inputs** and “avg” as **outputs**. We assign them to “feature_extractor” that is our functional built-from-scratch model that can extracts the features of our images (Tensors).
6. In order to extract the features we use the “.predict” method of the built model passing our independent variables (training images), and setting the **batch_size** to 64. We have also set the verbose equal to 1, to instruct the model to output its operations.

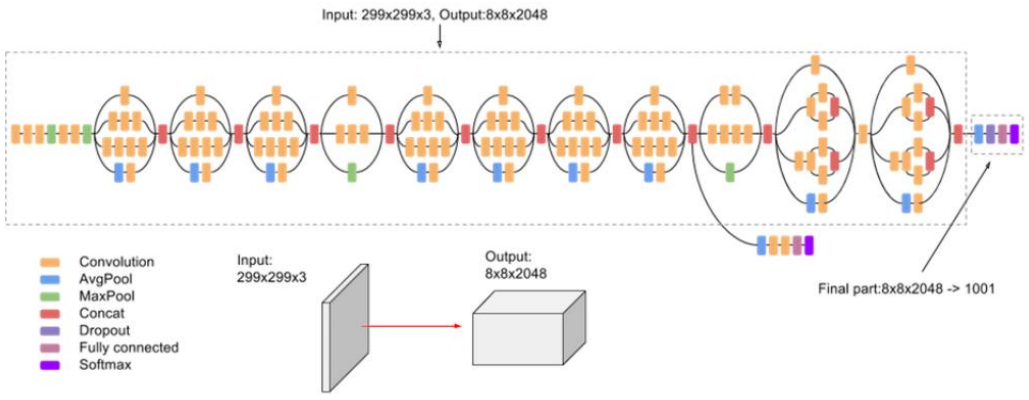
Which models did we use?

We used 4 different models, respectively **InceptionV3**, **Xception**, **NASNetLarge**, **InceptionResNetV2**. All of the models are imported from keras.applications and since each Keras Application expects a specific kind of input preprocessing. We have also imported “preprocess_input” for each model separately.

In the following sections, we explain each of these models and their functionality.

InceptionV3

Inception V3 is a widely used image recognition model that is the culmination of many ideas developed by multiple researchers over the years. It is the third edition of Google's Inception **Convolutional Neural Network**. Just as **ImageNet** can be thought of as a database of classified visual objects, Inception helps classification of objects in the world of computer vision. The model itself is made up of symmetric and asymmetric building blocks and its Loss is computed via **Softmax**. The default input is (299, 299, 3).



Inception V3 is from the Inception family that makes several improvements including using Label Smoothing, Factorized 7 x 7 convolutions (48 layers deep), and the use of an auxiliary classifer to propagate label information lower down the network.

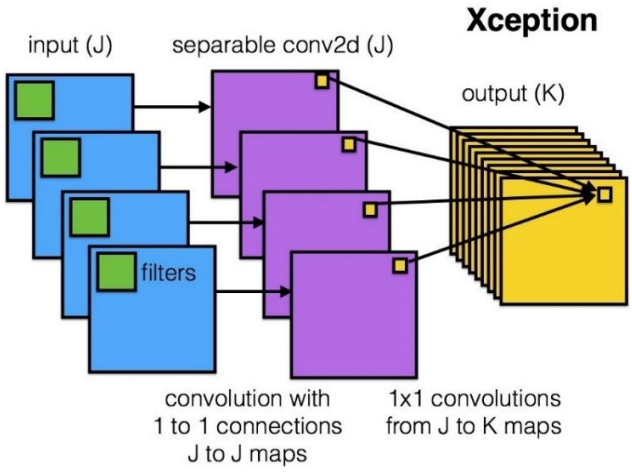
As we have explained in 3rd bullet point of the previous section, in the feature extraction phase, we have passed:

1. 'imagenet' to **weights**, since we wanted to use pre-training on ImageNet and we didn't have the path to the weights files.
2. False to **include_top**, to not include the fully connected layer at the top, as the last layer of the network. This is simply because the fully connected layers at the end can only take fixed size inputs, which has been previously defined by the input shape and all processing in the convolutional layers. Any change to the input shape will change the shape of the input to the fully connected layers, making the weights incompatible, since we are using a different size of input.
3. "input_size" to **input_shape**. That can be written with 2 formats, channel first or channel last.

Xception

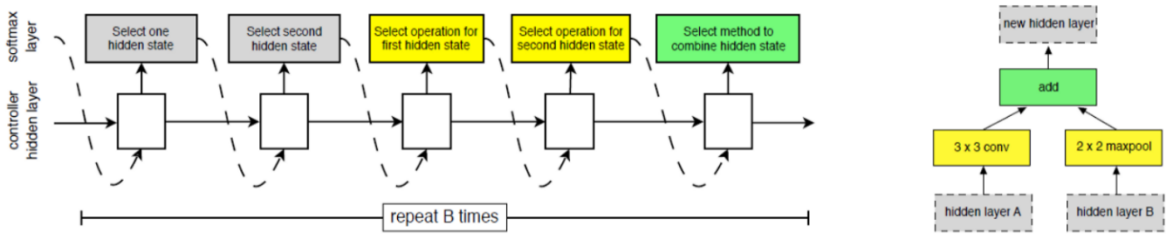
The model is a novel deep convolutional neural network architecture inspired by Inception, where Inception modules have been replaced with depth wise separable convolutions. Xception relies solely on depth wise separable convolution layers, however, this architecture (Xception) slightly outperforms Inception V3 on the ImageNet dataset (which Inception V3 was designed for), and significantly outperforms Inception V3 on a larger image classification dataset comprising 350 million images and 17,000 classes. Since the Xception architecture has the same number of parameters as Inception V3, the performance gains are not due to increased capacity but rather to a more efficient use of model parameters.

The Xception architecture has 36 convolutional layers (6x6) forming the feature extraction base of the network. In our experimental evaluation we will exclusively investigate image classification and therefore our convolutional base will be followed by a logistic regression layer. The default input is (299, 299, 3)



NASNetLarge

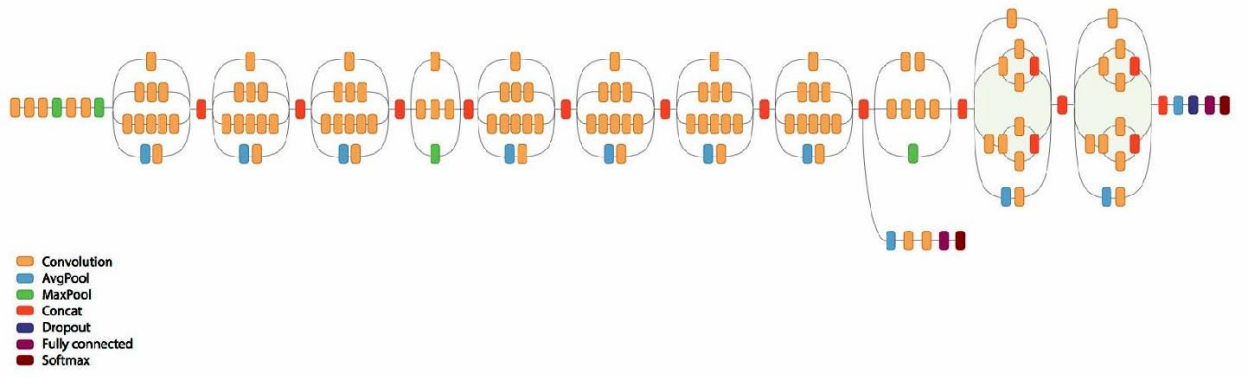
NASNetLarge (Neural Architecture Search Network - NASNet) is a convolutional neural network that is trained on more than a million images from the ImageNet database. The network can classify images into 1000 object categories. As a result, the network has learned rich feature representations for a wide range of images. The default input is (331, 331, 3)



Controller model architecture for recursively constructing one block of a convolutional cell

InceptionResNetV2

Inception-ResNet-v2 is a convolutional neural network that is trained on more than a million images from the ImageNet database. The network is 164 layers deep and can classify images into 1000 object. As a result, the network has learned rich feature representations for a wide range of images. The model has been built on the Inception family of architectures but incorporates residual connections (replacing the filter concatenation stage of the Inception architecture). Residual connections are a type of skip-connection that learn residual functions with reference to the layer inputs, instead of learning unreferenced functions.



After utilizing our custom function “get_features” to extract the features of our independent variables using all these 4 models and storing them in 4 separate variables. We have used **concatenate** from NumPy library to join the different extracted features into one single variable called “final_features”, that its shape is equal to (10222, 9664) that are the number of images and features, respectively.

We have set our callback using EarlyStopping that monitors validation loss for 3 steps.

How to build the model?

We have used the same method as the previous model, in which we have utilized Sequential from keras.models to group the linear layers of the model and used the following arguments:

1. Using InputLayer from tf.keras.layers as graph of layers to pass the shape of input features (9664,) to be used as an entry point in neural networks.
2. We have used the Dropout layer from tf.keras.layers that randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Since we have more than 10,000 images and using 4 different models to find features and extracted huge number of features, we have set the Dropout to 0.7.
3. Using tf.keras.layers.Dense by assigning the "number of our classes" (i.e. 120) to "unit".

For ".compile" method of our model and in order to introduce the variables for compiling and setting the desired configuration of the model for training, we have used the same arguments as the previous model which are:

- optimizer='adam'
- loss='categorical_crossentropy',
- metrics=['accuracy']

Checking the model summary using **model.summary()**:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dropout (Dropout)            (None, 9664)              0
_________________________________________________________________
dense (Dense)                (None, 120)               1159800
=================================================================
Total params: 1,159,800
Trainable params: 1,159,800
Non-trainable params: 0
```

Since we have added all the layers, we don't have any non-trainable parameters.

How to train the model?

We have fitted the model using the following arguments:

- "final_features" as our independent variables or features of the model
- "y" as our target, that are the boolean one hot arrays
- Number of epochs equal to 60
- Number of batch sizes equal to 128
- Setting validation set (10% of total data)
- Setting our callback using the EarlyStopping function and TensorBorad we have defined above.

Results

With this approach, we have expected a significant improvement in our validation accuracy. After the 1st epoch finished, the results were shocking, 0.9453 as our validation accuracy and they remained almost high after our callback stopped the learning phase on the 11th epoch, and the results were as following:

- Training Loss: 0.0411
- Training Accuracy: 0.9871
- Validation Loss: 0.1827
- Validation Accuracy: 0.9443

Models comparison

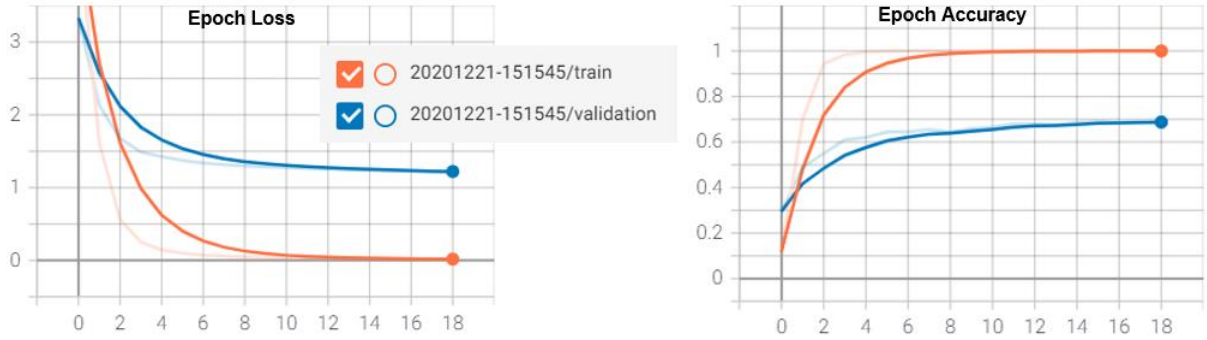
As we have anticipated before applying this new Functional method, we could increase our accuracy over validation set remarkably. This comparison suggests that for the relatively huge dataset (+10K images), implementing only one model is not recommended, specially having 120 different labels to classify.

Prediction and Submission on Kaggle

Doing all the preprocessing steps, this time for "test" images, extracting their features using our models, concatenating the features and making prediction using ".predict" method on our trained model, we have acquired the desired prediction labels. We have created a data frame (for each approach) with the structure that was expected on Kaggle submission section. We got a score of 0.88653 with the Linear approach (first notebook), and 0.18703 with the DNN approach (second notebook). The latter submission placed us among top 250 teams.

Graphs and Figures

Looking at the graphs that has been generated by TensorBoard, we observe that our model after 6th and 7th epoch gets completely overfitted on training set, even though the validation didn't change at all. Therefore, we see that both "accuracy" and "loss" didn't change significantly after the 6th epoch.



As we have expected, the Functional DNN model performed much better on validation data. By looking at the graphs, we see that the "accuracy" level on the training set keeps improving up until the last epoch, also looking at the "loss" values on the training set, we observe that it continues decreasing with the same pace.

On the other hand, we notice from the plots below that the "loss" and "accuracy" levels on the validation set, maintain very good results (accuracy = 0.94, loss = 0.19) from the beginning until the end.