Hyperparameters in Keras

quantra

Table of Contents

In the previous unit, you have learnt about the various layers in a deep neural network. In this document we will understand the various hyper-parameters for each of these layers. In this document, we also discuss a other features that are available in Keras while training any model.

Dense Layer

First let us look at the dense layer. Every Dense layer in a deep learning model implements the following operation:

output = activation(dot(input, kernel) + bias)

where activation is the element-wise activation function (tanh, sigmoid etc) passed as the activation argument, kernel is a weights matrix created by the layer and implemented on every corresponding neuron in the layer, and the bias is specific to every layer, and it is a constant value that is added to the weights vector.

Syntax:

```
keras.layers.Dense(units, activation=None, use_bias=True,
kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None,
bias_regularizer=None, activity_regularizer=None, kernel_constraint=None,
bias_constraint=None)
```

A dense layer has the following hyper parameters:

- units: This is a Positive integer, representing the number of neurons in that layer, and the dimensionality of the output space.
- activation: This will signify if an activation function is to be used. If you don't specify anything, a linear activation is used: a(x) = x. We will look at the various activation functions in a later unit.
- use_bias: Boolean, whether the layers should use the bias vector.

- kernel_initializer*: This a function that will help in Initializing the kernel/weights matrix for a layer
- bias_initializer*: This a function that will help in Initializing the bias vector.
- kernel_regularizer*: Regularizer function applied to the kernel weights matrix. Regularizers apply penalties on layer parameters or layer activity during optimization. These penalties are incorporated in the loss function that the network optimizes. In the upcoming units, we will learn how to plot the model's loss to visualize it.
- bias_regularizer*: This is like the kernel regularizer function but is used to set the values of the bias vector only.
- activity_regularizer*: This is similar to the kernel regularizer function but is applied to the output of a layer (or the activation).

Constraints:

A constraint is a limitation on one of the parameters that we want to train/monitor. Constraining the weight matrix directly is a kind of regularization used in deep learning. For example, if you use the maxnorm(m) function, where m is a positive number. It will scale your whole weight matrix by a factor that will reduce the square root of the sum of squares of the weights, also called the norm of the weight matrix. This constraint technique works well in combination with a dropout layer.

- kernel_constraint*: Constraint function applied to the kernel/weights matrix.
- bias_constraint*: Constraint function applied to the bias vector.

Activation Layer

We can declare the activation hyper-parameter discussed earlier as a separate layer. This layer applies an activation function (ex: relu, tanh) to the output of the previous layer.

Syntax:

```
keras.layers.Activation(activation)
```

It has a single hyper-parameter.

- activation: name of activation function to used

Dropout Layer

Dropout layers can be added after the activation function layer. The dropout layer has a single hyperparameter, which signifies the ratio of nodes to be dropped in a particular layer.

Syntax:

```
keras.layers.Dropout(rate, noise_shape=None, seed=None)
```

It has a single hyper-parameter.

- dropout: a float value between 0 and 1. Signifies the fraction of the units to be dropped.

Features in Keras

A class of functions called callbacks are used to accomplish a set of tasks, such as saving a model, monitoring the loss function, etc. A callback is used during the training phase of the model. For example, you can use callbacks to get a view on the statistics of the model, such as accuracy or loss, during training. You can pass a list of callbacks* (as the keyword

argument callbacks) to the .fit() method of the Sequential or Model classes. The relevant methods of the callbacks will then be called at each stage of the training. Let us understand some of the important callback functions.

ModelCheckpoint

This function helps us to save the model after every epoch.

Syntax:

```
keras.callbacks.ModelCheckpoint(filepath, monitor='val_loss', verbose=0,
save_best_only=False, save_weights_only=False, mode='auto', period=1)
```

It has the following parameters.

- filepath: It takes a string value, specifying the path on the local machine where the model will be saved.
- Monitor: It takes a string value. Specifies the quantity to monitor, such as the loss or accuracy of the model for every epoch.
- verbose: It takes a integer value. Specifies the verbosity mode, and can take the values 0 or 1. For a value 0, there won't be any update until the training is completed. For a value of 1, the monitored metric's value will be printed for every epoch.
- save_best_only: It is a boolean value. If save_best_only=True, the latest best model according to the quantity monitored will not be overwritten.
- mode: It takes a string value. It can be one of these three values {auto, min, max}. If save_best_only=True, the decision to overwrite the current save file is made based on either the maximization or the minimization of the monitored quantity. If we are monitoring the accuracy metric, then this should be max. Conversely if we

are monitoring the loss, then this should be min, etc. In auto mode, the direction is automatically inferred from the name of the monitored quantity.

- save_weights_only: It is a boolean value. If set to True, then only the model's weights will be saved (model.save_weights(filepath)), else the full model is saved (model.save(filepath)).
- period: It is a positive integer value. Specifies the interval (number of epochs) between checkpoints.

EarlyStopping

Early stopping attempts to remove the need to manually set the epoch value while training a model. It also helps us reduce the amount of time needed to train a model, as it stops the training process once the monitored quantity stops improving beyond a threshold value. This can stop the network from overfitting like a dropout layer, but at the same time reducing the train time.

Syntax:

```
keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0, patience=0, verbose=0, mode='auto', baseline=None)
```

EarlyStopping function as the following parameters:

- monitor: It is a string. Specifies the quantity to be monitored. For example: ‘loss’, or ‘accuracy’
- min_delta: It is a positive float value. Represents the minimum change in the monitored quantity needed to qualify as an improvement, For example: An absolute change of less than min_delta=0.1 in the accuracy will count as no improvement.

- patience: It is a positive integer. Represents the number of epochs with no improvement after which training will be stopped.
- verbose: Same as that of a ModelCheckpoint.
- mode: Same as that of a ModelCheckpoint.
- baseline: It is a float value. It represents the threshold value for the monitored quantity to enforce the early stopping. Training will stop if the model doesn't show improvement over the baseline. For example: If the baseline value is 0.6, and the parameter to be monitored is accuracy with a min_delta=0.1. So, if the accuracy of the model is less than 0.6, then the training continues even if the improvement in accuracy is less than min_delta specified. But as the accuracy increases above the 0.6, the early stopping criteria kicks in and stops the training if the there is no improvement in accuracy above the min_delta value.