Deep Learning Lecture 1 - Prevent overfitting in Keras

MA8701 General Statistical Methods

Thiago G. Martins, Department of Mathematical Sciences, NTNU

Spring 2019

Prevent overfitting

- When you see that the model's performance on the validation data begins to degrade, you've achieved overfitting.
- The next stage is to start regularizing and tuning the model, to get as close as possible to the ideal model that neither underfits nor overfits.
- These are the most common ways to prevent overfitting in neural networks:
  - Get more training data.
  - Reduce the complexity of the network.
  - Add weight regularization.
  - Add dropout.

Get more data

Getting more data than the number of effective parameters in the model is clearly an option to fight overfitting, although not always an easy one.

Reduce the complexity of the model

This can be accomplished by reducing the number of layers and/or reducing the number of hidden units in each layer.

Add weight regularization

In Keras, weight regularization is added by passing weight regularizer instances to layers as keyword arguments.

Adding L2 weight regularization to the model

```
model <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(10000),
              kernel_regularizer = regularizer_l2(0.001)) %>%
  layer_dense(units = 16, activation = "relu",
              kernel_regularizer = regularizer_l2(0.001)) %>%
  layer_dense(units = 1, activation = "sigmoid")
```

- `regularizer_l2(0.001)` means every coefficient in the weight matrix of the layer will add `0.001 * weight_coefficient_value` to the total loss of the network.
- Note that because this penalty is only added at training time, the loss for this network will be much higher at training time than at test time.

As an alternative to L2 regularization, you can use one of the following Keras weight regularizers

- `regularizer_l1(0.001)`
- `regularizer_l1_l2(l1 = 0.001, l2 = 0.001)`

Add dropout

- Dropout, applied to a layer, consists of randomly dropping out (setting to zero) a number of output features of the layer during training.
- The dropout rate is the fraction of the features that are zeroed out; it's usually set between 0.2 and 0.5.
- At test time, no units are dropped out; instead, the layer's output values are scaled down by a factor equal to the dropout rate, to balance for the fact that more units are active than at training time.
- The core idea is that introducing noise in the output values of a layer can break up happenstance patterns that aren't significant, which the network will start memorizing if no noise is present.

Example

Drop random units at training time (50% in this case):

```
layer_output <- layer_output * sample(0:1, length(layer_output),
                                      replace = TRUE)
```

At test time, use all units but scale their values down by 50%:

```
layer_output <- layer_output * 0.5
```

In practice, both operations can be implemented during training, leaving the output unchanged:

```
layer_output <- layer_output * sample(0:1, length(layer_output), replace = TRUE
layer_output <- layer_output / 0.5
```

Note that at training time we are scaling up instead of scaling down.

Dropout in Keras

In Keras, you can introduce dropout in a network via `layer_dropout`, which is applied to the output of the layer immediately before it:

```
layer_dropout(rate = 0.5)
```

```
model <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 1, activation = "sigmoid")
```

Reference material

This lecture note is based on (Chollet and Allaire 2018).

References

Chollet, F., and J. Allaire. 2018. *Deep Learning with R*. Manning Publications. https://books.google.no/books?id=xnIRtAEACAAJ.