The model performance data from Keras is in the following format:

# R code

library(wrapr)

df <- wrapr::build\_frame(

"val\_loss" , "val\_acc", "loss" , "acc" , "epoch" |

-0.377 , 0.8722 , -0.5067, 0.7852, 1 |

-0.2997 , 0.8895 , -0.3002, 0.904 , 2 |

-0.2964 , 0.8822 , -0.2166, 0.9303, 3 |

-0.2779 , 0.8899 , -0.1739, 0.9428, 4 |

-0.2843 , 0.8861 , -0.1411, 0.9545, 5 |

-0.312 , 0.8817 , -0.1136, 0.9656, 6 )

knitr::kable(df[1, , drop = FALSE])

| **val\_loss** | **val\_acc** | **loss** | **acc** | **epoch** |
| --- | --- | --- | --- | --- |
| -0.377 | 0.8722 | -0.5067 | 0.7852 | 1 |

And the form that would be easiest to use with ggplot2 would be the following:

# R code

pf <- wrapr::build\_frame(

"epoch" , "measure" , "training", "validation" |

1 , "minus binary cross entropy", -0.5067 , -0.377 |

1 , "accuracy" , 0.7852 , 0.8722 )

knitr::kable(pf)

| **epoch** | **measure** | **training** | **validation** |
| --- | --- | --- | --- |
| 1 | minus binary cross entropy | -0.5067 | -0.3770 |
| 1 | accuracy | 0.7852 | 0.8722 |

Here I show a cdata transform solution which we re-interpret as the following:

# R code

library(cdata)

# define the record shape we want by example

controlTable <- wrapr::qchar\_frame(

"measure" , "training", "validation" |

"minus binary cross entropy", loss , val\_loss |

"accuracy" , acc , val\_acc )

# use our example record shape to specify the record transform

transform <- rowrecs\_to\_blocks\_spec(

controlTable = controlTable,

recordKeys = 'epoch')

df %.>% transform

## epoch measure training validation

## 1 1 minus binary cross entropy -0.5067 -0.3770

## 2 1 accuracy 0.7852 0.8722

## 3 2 minus binary cross entropy -0.3002 -0.2997

## 4 2 accuracy 0.9040 0.8895

## 5 3 minus binary cross entropy -0.2166 -0.2964

## 6 3 accuracy 0.9303 0.8822

## 7 4 minus binary cross entropy -0.1739 -0.2779

## 8 4 accuracy 0.9428 0.8899

## 9 5 minus binary cross entropy -0.1411 -0.2843

## 10 5 accuracy 0.9545 0.8861

## 11 6 minus binary cross entropy -0.1136 -0.3120

## 12 6 accuracy 0.9656 0.8817

This simple data transform is in fact not a single pivot/un-pivot, as the result records spread data-values over multiple rows and multiple columns at the same time. We call the transform simple, because from a user point of view: it takes records of one form to another form (with the details left to the implementation).

In this note I would like to comment on some of the great advantages of using a data driven record transform specification.

Next: we can print the transformation and check if it matches our intent:

# R code

print(transform)

## {

## row\_record <- wrapr::qchar\_frame(

## "epoch" , "loss", "acc", "val\_loss", "val\_acc" |

## . , loss , acc , val\_loss , val\_acc )

## row\_keys <- c('epoch')

##

## # becomes

##

## block\_record <- wrapr::qchar\_frame(

## "epoch" , "measure" , "training", "validation" |

## . , "minus binary cross entropy", loss , val\_loss |

## . , "accuracy" , acc , val\_acc )

## block\_keys <- c('epoch', 'measure')

##

## # args: c(checkNames = TRUE, checkKeys = FALSE, strict = FALSE, allow\_rqdatatable = TRUE)

## }

The important point is that the transform is specified in data (not code):

# R code

str(transform)

## List of 7

## $ controlTable :'data.frame': 2 obs. of 3 variables:

## ..$ measure : chr [1:2] "minus binary cross entropy" "accuracy"

## ..$ training : chr [1:2] "loss" "acc"

## ..$ validation: chr [1:2] "val\_loss" "val\_acc"

## $ recordKeys : chr "epoch"

## $ controlTableKeys : chr "measure"

## $ checkNames : logi TRUE

## $ checkKeys : logi FALSE

## $ strict : logi FALSE

## $ allow\_rqdatatable: logi TRUE

## - attr(\*, "class")= chr "rowrecs\_to\_blocks\_spec"

Because the transform is data (not code), it is easy to share with other systems: such as SQL or Python/Pandas.

To show this we will first convert the transform specification into YAML for transport.

# R code

library(yaml)

yaml\_str <- transform %.>%

convert\_cdata\_spec\_to\_yaml(.) %.>%

yaml::as.yaml(.)

cat(yaml\_str)

## blocks\_out:

## record\_keys: epoch

## control\_table\_keys: measure

## control\_table:

## measure:

## - minus binary cross entropy

## - accuracy

## training:

## - loss

## - acc

## validation:

## - val\_loss

## - val\_acc

We can then import this structure into Python.

# R code

library(reticulate)

use\_condaenv("aiAcademy") # our Python environment, yours will be different

The transported operator can then be used in Python.

# Python code

import yaml

import pandas

import data\_algebra

from data\_algebra.cdata\_impl import record\_map\_from\_simple\_obj

record\_map = record\_map\_from\_simple\_obj(yaml.safe\_load(r.yaml\_str))

print(record\_map)

## Transform row records of the form:

## record\_keys: ['epoch']

## ['epoch', 'loss', 'acc', 'val\_loss', 'val\_acc']

## to block records of structure:

## RecordSpecification

## record\_keys: ['epoch']

## control\_table\_keys: ['measure']

## control\_table:

## measure training validation

## 0 minus binary cross entropy loss val\_loss

## 1 accuracy acc val\_acc

# Python code

print(r.df)

## val\_loss val\_acc loss acc epoch

## 0 -0.3770 0.8722 -0.5067 0.7852 1.0

## 1 -0.2997 0.8895 -0.3002 0.9040 2.0

## 2 -0.2964 0.8822 -0.2166 0.9303 3.0

## 3 -0.2779 0.8899 -0.1739 0.9428 4.0

## 4 -0.2843 0.8861 -0.1411 0.9545 5.0

## 5 -0.3120 0.8817 -0.1136 0.9656 6.0

# Python code

res = record\_map.transform(r.df)

print(res)

## epoch measure training validation

## 0 1.0 accuracy 0.7852 0.8722

## 1 1.0 minus binary cross entropy -0.5067 -0.3770

## 2 2.0 accuracy 0.9040 0.8895

## 3 2.0 minus binary cross entropy -0.3002 -0.2997

## 4 3.0 accuracy 0.9303 0.8822

## 5 3.0 minus binary cross entropy -0.2166 -0.2964

## 6 4.0 accuracy 0.9428 0.8899

## 7 4.0 minus binary cross entropy -0.1739 -0.2779

## 8 5.0 accuracy 0.9545 0.8861

## 9 5.0 minus binary cross entropy -0.1411 -0.2843

## 10 6.0 accuracy 0.9656 0.8817

## 11 6.0 minus binary cross entropy -0.1136 -0.3120

We can even convert the transform to SQL (either in R directly or in Python directly).

# Python code

from data\_algebra.SQLite import SQLiteModel

from data\_algebra.data\_ops import \*

db\_model = SQLiteModel()

ops = TableDescription(

'keras\_frame',

["val\_loss", "val\_acc", "loss", "acc", "epoch"]). \

convert\_records(record\_map)

sql\_str = ops.to\_sql(db\_model, pretty=True)

print(sql\_str)

## SELECT a."epoch" AS "epoch",

## b."measure" AS "measure",

## CASE

## WHEN b."training" = 'loss' THEN a."loss"

## WHEN b."training" = 'acc' THEN a."acc"

## ELSE NULL

## END AS "training",

## CASE

## WHEN b."validation" = 'val\_loss' THEN a."val\_loss"

## WHEN b."validation" = 'val\_acc' THEN a."val\_acc"

## ELSE NULL

## END AS "validation"

## FROM ("keras\_frame") a

## CROSS JOIN ("cdata\_temp\_record") b

## ORDER BY a."epoch",

## b."measure"

The SQL code was generated from the transform specification. This was easy to implement as it is often simple to convert data to code (though it can be quite hard to translate code to data).And that is some of the power of using data to specify your data transforms.