# Data input

We work with the same data as before, vic\_elec from tsibbledata.

Compared to last time though, the dataset class has to change. While, previously, for each batch item the target (y) was a single value, it now is a vector, just like the input, x. And just like n\_timesteps was (and still is) used to specify the length of the input sequence, there is now a second parameter, n\_forecast, to configure target size.

In our example, n\_timesteps and n\_forecast are set to the same value, but there is no need for this to be the case. You could equally well train on week-long sequences and then forecast developments over a single day, or a month.

Apart from the fact that .getitem() now returns a vector for y as well as x, there is not much to be said about dataset creation. Here is the complete code to set up the data input pipeline:

n\_timesteps <- 7 \* 24 \* 2

n\_forecast <- 7 \* 24 \* 2

batch\_size <- 32

vic\_elec\_get\_year <- function(year, month = NULL) { vic\_elec %>%

filter(year(Date) == year, month(Date) == if (is.null(month)) month(Date) else month) %>%

as\_tibble() %>% select(Demand)

}

elec\_train <- vic\_elec\_get\_year(2012) %>% as.matrix() elec\_valid <- vic\_elec\_get\_year(2013) %>% as.matrix() elec\_test <- vic\_elec\_get\_year(2014, 1) %>% as.matrix()

train\_mean <- mean(elec\_train) train\_sd <- sd(elec\_train)

elec\_dataset <- dataset( name = "elec\_dataset",

initialize = function(x, n\_timesteps, n\_forecast, sample\_frac = 1) { self$n\_timesteps <- n\_timesteps

self$n\_forecast <- n\_forecast

self$x <- torch\_tensor((x - train\_mean) / train\_sd)

n <- length(self$x) - self$n\_timesteps - self$n\_forecast + 1 self$starts <- sort(sample.int(

n = n,

size = n \* sample\_frac

))

},

.getitem = function(i) { start <- self$starts[i]

end <- start + self$n\_timesteps - 1 pred\_length <- self$n\_forecast

list(

x = self$x[start:end],

y = self$x[(end + 1):(end + pred\_length)]$squeeze(2)

)

},

.length = function() { length(self$starts)

}

)

train\_ds <- elec\_dataset(elec\_train, n\_timesteps, n\_forecast, sample\_frac = 0.5)

train\_dl <- train\_ds %>% dataloader(batch\_size = batch\_size, shuffle = TRUE)

valid\_ds <- elec\_dataset(elec\_valid, n\_timesteps, n\_forecast, sample\_frac = 0.5)

valid\_dl <- valid\_ds %>% dataloader(batch\_size = batch\_size)

test\_ds <- elec\_dataset(elec\_test, n\_timesteps, n\_forecast) test\_dl <- test\_ds %>% dataloader(batch\_size = 1)

# Model

The model replaces the single linear layer that, in the previous post, had been tasked with outputting the final prediction, with a small network, complete with two linear layers and – optional – dropout.

In forward(), we first apply the RNN, and just like in the previous post, we make use of the outputs only; or more specifically, the output corresponding to the final time step.

model <- nn\_module(

initialize = function(type, input\_size, hidden\_size, linear\_size, output\_size,

num\_layers = 1, dropout = 0, linear\_dropout =

0) {

self$type <- type self$num\_layers <- num\_layers

self$linear\_dropout <- linear\_dropout

self$rnn <- if (self$type == "gru") { nn\_gru(

input\_size = input\_size, hidden\_size = hidden\_size, num\_layers = num\_layers, dropout = dropout, batch\_first = TRUE

)

} else { nn\_lstm(

input\_size = input\_size, hidden\_size = hidden\_size, num\_layers = num\_layers, dropout = dropout, batch\_first = TRUE

)

}

self$mlp <- nn\_sequential( nn\_linear(hidden\_size, linear\_size), nn\_relu(), nn\_dropout(linear\_dropout), nn\_linear(linear\_size, output\_size)

)

},

forward = function(x) {

x <- self$rnn(x) x[[1]][ ,-1, ..] %>%

self$mlp()

}

)

For model instantiation, we now have an additional configuration parameter, related to the amount of dropout between the two linear layers.

net <- model(

"gru", input\_size = 1, hidden\_size = 32, linear\_size = 512, output\_size = n\_forecast, linear\_dropout = 0

)

# training RNNs on the GPU currently prints a warning that may clutter # the console

# alternatively, use # device <- "cpu"

device <- torch\_device(if (cuda\_is\_available()) "cuda" else "cpu")

net <- net$to(device = device)

# Training

The training procedure is completely unchanged.

optimizer <- optim\_adam(net$parameters, lr = 0.001) num\_epochs <- 30

train\_batch <- function(b) {

optimizer$zero\_grad()

output <- net(b$x$to(device = device)) target <- b$y$to(device = device)

loss <- nnf\_mse\_loss(output, target) loss$backward()

optimizer$step()

loss$item()

}

valid\_batch <- function(b) {

output <- net(b$x$to(device = device)) target <- b$y$to(device = device)

loss <- nnf\_mse\_loss(output, target) loss$item()

}

for (epoch in 1:num\_epochs) { net$train()

train\_loss <- c()

coro::loop(for (b in train\_dl) { loss <-train\_batch(b)

train\_loss <- c(train\_loss, loss)

})

cat(sprintf("\nEpoch %d, training: loss: %3.5f \n", epoch, mean(train\_loss))

net$eval() valid\_loss <- c()

coro::loop(for (b in valid\_dl) {

loss <- valid\_batch(b)

valid\_loss <- c(valid\_loss, loss)

})

cat(sprintf("\nEpoch %d, validation: loss: %3.5f \n", epoch, mean(valid\_loss))

}

|  |  |  |  |
| --- | --- | --- | --- |
| #  # | Epoch | 1, | training: loss: 0.65737 |
| # | Epoch | 1, | validation: loss: 0.54586 |
| # |  |  |  |
| # | Epoch | 2, | training: loss: 0.43991 |
| # |  |  |  |
| # | Epoch | 2, | validation: loss: 0.50588 |
| # |  |  |  |
| # | Epoch | 3, | training: loss: 0.42161 |
| # |  |  |  |
| # | Epoch | 3, | validation: loss: 0.50031 |
| # |  |  |  |
| # | Epoch | 4, | training: loss: 0.41718 |
| # |  |  |  |
| # | Epoch | 4, | validation: loss: 0.48703 |
| # |  |  |  |
| # | Epoch | 5, | training: loss: 0.39498 |
| # |  |  |  |
| # | Epoch | 5, | validation: loss: 0.49572 |
| # |  |  |  |
| # | Epoch | 6, | training: loss: 0.38073 |
| # |  |  |  |
| # | Epoch | 6, | validation: loss: 0.46813 |
| # |  |  |  |
| # | Epoch | 7, | training: loss: 0.36472 |
| # |  |  |  |
| # | Epoch | 7, | validation: loss: 0.44957 |
| # |  |  |  |
| # | Epoch | 8, | training: loss: 0.35058 |
| # |  |  |  |
| # | Epoch | 8, | validation: loss: 0.44440 |
| # |  |  |  |
| # | Epoch | 9, | training: loss: 0.33880 |
| # |  |  |  |
| # | Epoch | 9, | validation: loss: 0.41995 |
| # |  |  |  |
| # | Epoch | 10, | training: loss: 0.32545 |
| # |  |  |  |
| # | Epoch | 10, | validation: loss: 0.42021 |
| # |  |  |  |
| # | Epoch | 11, | training: loss: 0.31347 |
| # |  |  |  |
| # | Epoch | 11, | validation: loss: 0.39514 |
| # |  |  |  |
| # | Epoch | 12, | training: loss: 0.29622 |

|  |  |  |  |
| --- | --- | --- | --- |
| # |  |  |  |
| # | Epoch | 12, | validation: loss: 0.38146 |
| # |  |  |  |
| # | Epoch | 13, | training: loss: 0.28006 |
| # |  |  |  |
| # | Epoch | 13, | validation: loss: 0.37754 |
| # |  |  |  |
| # | Epoch | 14, | training: loss: 0.27001 |
| # |  |  |  |
| # | Epoch | 14, | validation: loss: 0.36636 |
| # |  |  |  |
| # | Epoch | 15, | training: loss: 0.26191 |
| # |  |  |  |
| # | Epoch | 15, | validation: loss: 0.35338 |
| # |  |  |  |
| # | Epoch | 16, | training: loss: 0.25533 |
| # |  |  |  |
| # | Epoch | 16, | validation: loss: 0.35453 |
| # |  |  |  |
| # | Epoch | 17, | training: loss: 0.25085 |
| # |  |  |  |
| # | Epoch | 17, | validation: loss: 0.34521 |
| # |  |  |  |
| # | Epoch | 18, | training: loss: 0.24686 |
| # |  |  |  |
| # | Epoch | 18, | validation: loss: 0.35094 |
| # |  |  |  |
| # | Epoch | 19, | training: loss: 0.24159 |
| # |  |  |  |
| # | Epoch | 19, | validation: loss: 0.33776 |
| # |  |  |  |
| # | Epoch | 20, | training: loss: 0.23680 |
| # |  |  |  |
| # | Epoch | 20, | validation: loss: 0.33974 |
| # |  |  |  |
| # | Epoch | 21, | training: loss: 0.23070 |
| # |  |  |  |
| # | Epoch | 21, | validation: loss: 0.34069 |
| # |  |  |  |
| # | Epoch | 22, | training: loss: 0.22761 |
| # |  |  |  |
| # | Epoch | 22, | validation: loss: 0.33724 |
| # |  |  |  |
| # | Epoch | 23, | training: loss: 0.22390 |
| # |  |  |  |
| # | Epoch | 23, | validation: loss: 0.34013 |
| # |  |  |  |
| # | Epoch | 24, | training: loss: 0.22155 |
| # |  |  |  |
| # | Epoch | 24, | validation: loss: 0.33460 |
| # |  |  |  |
| # | Epoch | 25, | training: loss: 0.21820 |

|  |  |  |  |
| --- | --- | --- | --- |
| # |  |  |  |
| # | Epoch | 25, | validation: loss: 0.33755 |
| # |  |  |  |
| # | Epoch | 26, | training: loss: 0.22134 |
| # |  |  |  |
| # | Epoch | 26, | validation: loss: 0.33678 |
| # |  |  |  |
| # | Epoch | 27, | training: loss: 0.21061 |
| # |  |  |  |
| # | Epoch | 27, | validation: loss: 0.33108 |
| # |  |  |  |
| # | Epoch | 28, | training: loss: 0.20496 |
| # |  |  |  |
| # | Epoch | 28, | validation: loss: 0.32769 |
| # |  |  |  |
| # | Epoch | 29, | training: loss: 0.20223 |
| # |  |  |  |
| # | Epoch | 29, | validation: loss: 0.32969 |
| # |  |  |  |
| # | Epoch | 30, | training: loss: 0.20022 |
| # |  |  |  |
| # | Epoch | 30, | validation: loss: 0.33331 |

From the way loss decreases on the training set, we conclude that, yes, the model is learning something. It probably would continue improving for quite some epochs still. We do, however, see less of an improvement on the validation set.

Naturally, now we’re curious about test-set predictions. (Remember, for testing we’re choosing the “particularly hard” month of January, 2014 – particularly hard because of a heatwave that resulted in exceptionally high demand.)

# Evaluation

With no loop to be coded, evaluation now becomes pretty straightforward:

net$eval()

test\_preds <- vector(mode = "list", length = length(test\_dl)) i <- 1

coro::loop(for (b in test\_dl) {

input <- b$x

output <- net(input$to(device = device)) preds <- as.numeric(output)

test\_preds[[i]] <- preds i <<- i + 1

})

vic\_elec\_jan\_2014 <- vic\_elec %>%

filter(year(Date) == 2014, month(Date) == 1)

test\_pred1 <- test\_preds[[1]]

test\_pred1 <- c(rep(NA, n\_timesteps), test\_pred1, rep(NA, nrow(vic\_elec\_jan\_2014) - n\_timesteps - n\_forecast))

test\_pred2 <- test\_preds[[408]]

test\_pred2 <- c(rep(NA, n\_timesteps + 407), test\_pred2, rep(NA, nrow(vic\_elec\_jan\_2014) - 407 - n\_timesteps - n\_forecast))

test\_pred3 <- test\_preds[[817]]

test\_pred3 <- c(rep(NA, nrow(vic\_elec\_jan\_2014) - n\_forecast), test\_pred3)

preds\_ts <- vic\_elec\_jan\_2014 %>% select(Demand) %>%

add\_column(

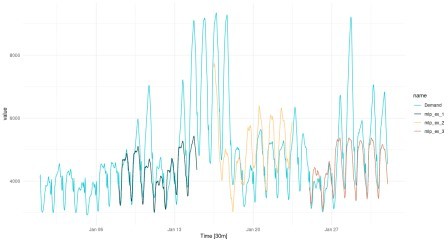
mlp\_ex\_1 = test\_pred1 \* train\_sd + train\_mean, mlp\_ex\_2 = test\_pred2 \* train\_sd + train\_mean, mlp\_ex\_3 = test\_pred3 \* train\_sd + train\_mean) %>%

pivot\_longer(-Time) %>% update\_tsibble(key = name)

preds\_ts %>% autoplot() +

scale\_colour\_manual(values = c("#08c5d1", "#00353f", "#ffbf66", "#d46f4d")) +

theme\_minimal()



(#fig:unnamed-chunk-6)One-week-ahead predictions for January, 2014.

Compare this to the forecast obtained by feeding back predictions. The demand profiles over the day look a lot more realistic now. How about the phases of extreme demand? Evidently, these are not reflected in the forecast, not any more than in the “loop technique”. In fact, the forecast allows for interesting insights into this model’s personality: Apparently, it really likes fluctuating around the mean – “prime” it with inputs that oscillate around a significantly higher level, and it will quickly shift back to its comfort zone.

# Discussion

Seeing how, above, we provided an option to use dropout inside the MLP, you may be wondering if this would help with forecasts on the test set. Turns out it did not, in my experiments. Maybe this is not so strange either: How, absent external cues (temperature), should the network know that high demand is coming up?

In our analysis, we can make an additional distinction. With the first week of predictions, what we see is a failure to anticipate something that *could not* reasonably have been anticipated (two, or two-and-a-half, say, days of exceptionally high demand). In the second, all the network would have had to do was stay at the current, elevated level. It will be interesting to see how this is handled by the architectures we discuss next.

Finally, an additional idea you may have had is – what if we used temperature as a second input variable? As a matter of fact, training performance indeed improved, but no performance impact was observed on the validation and test sets. Still, you may find the code useful – it is easily extended to datasets with more predictors. Therefore, we reproduce it in the appendix.

Thanks for reading!

# Appendix

# Data input code modified to accommodate two predictors

n\_timesteps <- 7 \* 24 \* 2

n\_forecast <- 7 \* 24 \* 2

vic\_elec\_get\_year <- function(year, month = NULL) { vic\_elec %>%

filter(year(Date) == year, month(Date) == if (is.null(month)) month(Date) else month) %>%

as\_tibble() %>% select(Demand, Temperature)

}

elec\_train <- vic\_elec\_get\_year(2012) %>% as.matrix() elec\_valid <- vic\_elec\_get\_year(2013) %>% as.matrix() elec\_test <- vic\_elec\_get\_year(2014, 1) %>% as.matrix()

train\_mean\_demand <- mean(elec\_train[ , 1]) train\_sd\_demand <- sd(elec\_train[ , 1])

train\_mean\_temp <- mean(elec\_train[ , 2]) train\_sd\_temp <- sd(elec\_train[ , 2])

elec\_dataset <- dataset( name = "elec\_dataset",

initialize = function(data, n\_timesteps, n\_forecast, sample\_frac = 1)

{

demand <- (data[ , 1] - train\_mean\_demand) / train\_sd\_demand temp <- (data[ , 2] - train\_mean\_temp) / train\_sd\_temp

self$x <- cbind(demand, temp) %>% torch\_tensor()

self$n\_timesteps <- n\_timesteps self$n\_forecast <- n\_forecast

n <- nrow(self$x) - self$n\_timesteps - self$n\_forecast + 1 self$starts <- sort(sample.int(

n = n,

size = n \* sample\_frac

))

},

.getitem = function(i) { start <- self$starts[i]

end <- start + self$n\_timesteps - 1 pred\_length <- self$n\_forecast

list(

x = self$x[start:end, ],

y = self$x[(end + 1):(end + pred\_length), 1]

)

},

.length = function() { length(self$starts)

}

)

### rest identical to single-predictor code above