# Data input

As before, we work with `vic_elec`, but this time, we partly deviate from the way we used to employ it. With the original, bi-hourly dataset, training the current model takes a long time, longer than readers will want to wait when experimenting. So instead, we aggregate observations by day. In order to have enough data, we train on years 2012 and 2013, reserving 2014 for validation as well as post-training inspection.

```
vic_elec_daily <- vic_elec %>%
  select(Time, Demand) %>%
  index_by(Date = date(Time)) %>%
  summarise(
    Demand = sum(Demand) / 1e3)

elec_train <- vic_elec_daily %>%
  filter(year(Date) %in% c(2012, 2013)) %>%
  as_tibble() %>%
  select(Demand) %>%
  as.matrix()

elec_valid <- vic_elec_daily %>%
  filter(year(Date) == 2014) %>%
  as_tibble() %>%
  select(Demand) %>%
  as.matrix()

elec_test <- vic_elec_daily %>%
  filter(year(Date) %in% c(2014), month(Date) %in% 1:4) %>%
  as_tibble() %>%
  select(Demand) %>%
  as.matrix()

train_mean <- mean(elec_train)
train_sd <- sd(elec_train)
```

We'll attempt to forecast demand up to fourteen days ahead. How long, then, should be the input sequences? This is a matter of experimentation; all the more so now that we're adding in the attention mechanism. (I suspect that it might not handle very long sequences so well).

Below, we go with fourteen days for input length, too, but that may not necessarily be the best possible choice for this series.

```
n_timesteps <- 7 * 2
n_forecast <- 7 * 2

elec_dataset <- dataset(
  name = "elec_dataset",

  initialize = function(x, n_timesteps, sample_frac = 1) {

    self$n_timesteps <- n_timesteps
```

```
    self$x <- torch_tensor((x - train_mean) / train_sd)

    n <- length(self$x) - self$n_timesteps - 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
    lag <- 1

    list(
      x = self$x[start:end],
      y = self$x[(start+lag):(end+lag)]$squeeze(2)
    )

  },

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

batch_size <- 32

train_ds <- elec_dataset(elec_train, n_timesteps, sample_frac = 0.5)
train_dl <- train_ds %>% dataloader(batch_size = batch_size, shuffle =
TRUE)

valid_ds <- elec_dataset(elec_valid, n_timesteps, sample_frac = 0.5)
valid_dl <- valid_ds %>% dataloader(batch_size = batch_size)

test_ds <- elec_dataset(elec_test, n_timesteps)
test_dl <- test_ds %>% dataloader(batch_size = 1)
```

# Model

Model-wise, we again encounter the three *modules* familiar from the previous post: encoder, decoder, and top-level seq2seq module. However, there is an additional component: the *attention* module, used by the decoder to obtain *attention weights*.

### Encoder

The encoder still works the same way. It wraps an RNN, and returns the final state.

```
encoder_module <- nn_module(
```

```
  initialize = function(type, input_size, hidden_size, num_layers = 1,
dropout = 0) {

    self$type <- type

    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
      )
    }

  },

  forward = function(x) {

    x <- self$rnn(x)

    # return last states for all layers
    # per layer, a single tensor for GRU, a list of 2 tensors for LSTM
    x <- x[[2]]
    x

  }
)
```

## Attention module

In basic seq2seq, whenever it had to generate a new value, the decoder took into account two things: its prior state, and the previous output generated. In an attention-enriched setup, the decoder additionally receives the complete output from the encoder. In deciding what subset of that output should matter, it gets help from a new agent, the attention module.

This, then, is the attention module's raison d'être: Given current decoder state and well as complete encoder outputs, obtain a weighting of those outputs indicative of how relevant they are to what the decoder is currently up to. This procedure results in the so-called *attention weights*: a normalized score, for each time step in the encoding, that quantify their respective importance.

Attention may be implemented in a number of different ways. Here, we show two implementation options, one additive, and one multiplicative.

**Additive attention**

In additive attention, encoder outputs and decoder state are commonly either added or concatenated (we choose to do the latter, below). The resulting tensor is run through a linear layer, and a softmax is applied for normalization.

```
attention_module_additive <- nn_module(

  initialize = function(hidden_dim, attention_size) {

    self$attention <- nn_linear(2 * hidden_dim, attention_size)

  },

  forward = function(state, encoder_outputs) {

    # function argument shapes
    # encoder_outputs: (bs, timesteps, hidden_dim)
    # state: (1, bs, hidden_dim)

    # multiplex state to allow for concatenation (dimensions 1 and 2
must agree)
    seq_len <- dim(encoder_outputs)[2]
    # resulting shape: (bs, timesteps, hidden_dim)
    state_rep <- state$permute(c(2, 1, 3))$repeat_interleave(seq_len,
2)

    # concatenate along feature dimension
    concat <- torch_cat(list(state_rep, encoder_outputs), dim = 3)

    # run through linear layer with tanh
    # resulting shape: (bs, timesteps, attention_size)
    scores <- self$attention(concat) %>%
      torch_tanh()

    # sum over attention dimension and normalize
    # resulting shape: (bs, timesteps)
    attention_weights <- scores %>%
      torch_sum(dim = 3) %>%
      nnf_softmax(dim = 2)

    # a normalized score for every source token
    attention_weights
  }
)
```

**Multiplicative attention**

In multiplicative attention, scores are obtained by computing dot products between decoder state and all of the encoder outputs. Here too, a softmax is then used for normalization.

```
attention_module_multiplicative <- nn_module(
```

```
  initialize = function() {

    NULL

  },

  forward = function(state, encoder_outputs) {

    # function argument shapes
    # encoder_outputs: (bs, timesteps, hidden_dim)
    # state: (1, bs, hidden_dim)

    # allow for matrix multiplication with encoder_outputs
    state <- state$permute(c(2, 3, 1))

    # prepare for scaling by number of features
    d <- torch_tensor(dim(encoder_outputs)[3], dtype = torch_float())

    # scaled dot products between state and outputs
    # resulting shape: (bs, timesteps, 1)
    scores <- torch_bmm(encoder_outputs, state) %>%
      torch_div(torch_sqrt(d))

    # normalize
    # resulting shape: (bs, timesteps)
    attention_weights <- scores$squeeze(3) %>%
      nnf_softmax(dim = 2)

    # a normalized score for every source token
    attention_weights
  }
)
```

## Decoder

Once attention weights have been computed, their actual application is handled by the decoder. Concretely, the method in question, `weighted_encoder_outputs()`, computes a product of weights and encoder outputs, making sure that each output will have appropriate impact.

The rest of the action then happens in `forward()`. A concatenation of weighted encoder outputs (often called "context") and current input is run through an RNN. Then, an ensemble of RNN output, context, and input is passed to an MLP. Finally, both RNN state and current prediction are returned.

```
decoder_module <- nn_module(

  initialize = function(type, input_size, hidden_size, attention_type,
attention_size = 8, num_layers = 1) {

    self$type <- type
```

```
    self$rnn <- if (self$type == "gru") {
      nn_gru(
        input_size = input_size,
        hidden_size = hidden_size,
        num_layers = num_layers,
        batch_first = TRUE
      )
    } else {
      nn_lstm(
        input_size = input_size,
        hidden_size = hidden_size,
        num_layers = num_layers,
        batch_first = TRUE
      )
    }

    self$linear <- nn_linear(2 * hidden_size + 1, 1)

    self$attention <- if (attention_type == "multiplicative")
attention_module_multiplicative()
      else attention_module_additive(hidden_size, attention_size)

  },

  weighted_encoder_outputs = function(state, encoder_outputs) {

    # encoder_outputs is (bs, timesteps, hidden_dim)
    # state is (1, bs, hidden_dim)
    # resulting shape: (bs * timesteps)
    attention_weights <- self$attention(state, encoder_outputs)

    # resulting shape: (bs, 1, seq_len)
    attention_weights <- attention_weights$unsqueeze(2)

    # resulting shape: (bs, 1, hidden_size)
    weighted_encoder_outputs <- torch_bmm(attention_weights,
encoder_outputs)

    weighted_encoder_outputs

  },

  forward = function(x, state, encoder_outputs) {

    # encoder_outputs is (bs, timesteps, hidden_dim)
    # state is (1, bs, hidden_dim)

    # resulting shape: (bs, 1, hidden_size)
    context <- self$weighted_encoder_outputs(state, encoder_outputs)

    # concatenate input and context
    # NOTE: this repeating is done to compensate for the absence of an
```

```
embedding module
    # that, in NLP, would give x a higher proportion in the
concatenation
    x_rep <- x$repeat_interleave(dim(context)[3], 3)
    rnn_input <- torch_cat(list(x_rep, context), dim = 3)

    # resulting shapes: (bs, 1, hidden_size) and (1, bs, hidden_size)
    rnn_out <- self$rnn(rnn_input, state)
    rnn_output <- rnn_out[[1]]
    next_hidden <- rnn_out[[2]]

    mlp_input <- torch_cat(list(rnn_output$squeeze(2),
context$squeeze(2), x$squeeze(2)), dim = 2)

    output <- self$linear(mlp_input)

    # shapes: (bs, 1) and (1, bs, hidden_size)
    list(output, next_hidden)
  }

)
```

### `seq2seq` module

The `seq2seq` module is basically unchanged (apart from the fact that now, it allows for attention module configuration). For a detailed explanation of what happens here, please consult the previous post.

```
seq2seq_module <- nn_module(

  initialize = function(type, input_size, hidden_size, attention_type,
attention_size, n_forecast,
                        num_layers = 1, encoder_dropout = 0) {

    self$encoder <- encoder_module(type = type, input_size =
input_size, hidden_size = hidden_size,
                                   num_layers, encoder_dropout)
    self$decoder <- decoder_module(type = type, input_size = 2 *
hidden_size, hidden_size = hidden_size,
                                   attention_type = attention_type,
attention_size = attention_size, num_layers)
    self$n_forecast <- n_forecast

  },

  forward = function(x, y, teacher_forcing_ratio) {

    outputs <- torch_zeros(dim(x)[1], self$n_forecast)$to(device =
device)
    encoded <- self$encoder(x)
    encoder_outputs <- encoded[[1]]
    hidden <- encoded[[2]]
```

```
    # list of (batch_size, 1), (1, batch_size, hidden_size)
    out <- self$decoder(x[ , n_timesteps, , drop = FALSE], hidden,
encoder_outputs)
    # (batch_size, 1)
    pred <- out[[1]]
    # (1, batch_size, hidden_size)
    state <- out[[2]]
    outputs[ , 1] <- pred$squeeze(2)

    for (t in 2:self$n_forecast) {

      teacher_forcing <- runif(1) < teacher_forcing_ratio
      input <- if (teacher_forcing == TRUE) pred$unsqueeze(3) else y[ ,
t - 1]
      out <- self$decoder(pred$unsqueeze(3), state, encoder_outputs)
      pred <- out[[1]]
      state <- out[[2]]
      outputs[ , t] <- pred$squeeze(2)

    }

    outputs
  }

)
```

When instantiating the top-level model, we now have an additional choice: that between additive and multiplicative attention. In the “accuracy” sense of performance, my tests did not show any differences. However, the multiplicative variant is a lot faster.

```
net <- seq2seq_module("gru", input_size = 1, hidden_size = 32,
attention_type = "multiplicative",
                      attention_size = 8, n_forecast = n_timesteps)

# training RNNs on the GPU currently prints a warning that may clutter
# the console
# see https://github.com/mlverse/torch/issues/461
# alternatively, use
# device <- "cpu"
device <- torch_device(if (cuda_is_available()) "cuda" else "cpu")

net <- net$to(device = device)
```

# Training

Just like last time, in model training, we get to choose the degree of teacher forcing. Below, we go with a fraction of 0.0, that is, no forcing at all.

```
optimizer <- optim_adam(net$parameters, lr = 0.001)

num_epochs <- 100

train_batch <- function(b, teacher_forcing_ratio) {
```

```
  optimizer$zero_grad()
  output <- net(b$x$to(device = device), b$y$to(device = device),
teacher_forcing_ratio)
  target <- b$y$to(device = device)

  loss <- nnf_mse_loss(output, target)
  loss$backward()
  optimizer$step()

  loss$item()

}

valid_batch <- function(b, teacher_forcing_ratio = 0) {

  output <- net(b$x$to(device = device), b$y$to(device = device),
teacher_forcing_ratio)
  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, teacher_forcing_ratio = 0.3)
    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: 0.83752
# Epoch 1, validation: loss: 0.83167
```

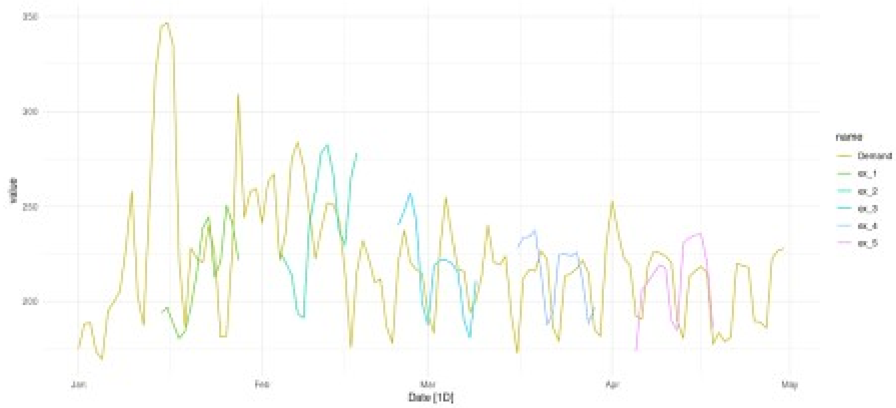
```
# Epoch 2, training: loss: 0.72803
# Epoch 2, validation: loss: 0.80804

# ...
# ...

# Epoch 99, training: loss: 0.10385
# Epoch 99, validation: loss: 0.21259

# Epoch 100, training: loss: 0.10396
# Epoch 100, validation: loss: 0.20975
```

# Evaluation

For visual inspection, we pick a few forecasts from the test set.

```
net$eval()

test_preds <- vector(mode = "list", length = length(test_dl))

i <- 1

vic_elec_test <- vic_elec_daily %>%
  filter(year(Date) == 2014, month(Date) %in% 1:4)


coro::loop(for (b in test_dl) {

  input <- b$x
  output <- net(b$x$to(device = device), b$y$to(device = device),
teacher_forcing_ratio = 0)
  preds <- as.numeric(output)

  test_preds[[i]] <- preds
  i <<- i + 1

})

test_pred1 <- test_preds[[1]]
test_pred1 <- c(rep(NA, n_timesteps), test_pred1, rep(NA,
nrow(vic_elec_test) - n_timesteps - n_forecast))

test_pred2 <- test_preds[[21]]
test_pred2 <- c(rep(NA, n_timesteps + 20), test_pred2, rep(NA,
nrow(vic_elec_test) - 20 - n_timesteps - n_forecast))

test_pred3 <- test_preds[[41]]
test_pred3 <- c(rep(NA, n_timesteps + 40), test_pred3, rep(NA,
nrow(vic_elec_test) - 40 - n_timesteps - n_forecast))

test_pred4 <- test_preds[[61]]
```

```
test_pred4 <- c(rep(NA, n_timesteps + 60), test_pred4, rep(NA,
nrow(vic_elec_test) - 60 - n_timesteps - n_forecast))

test_pred5 <- test_preds[[81]]
test_pred5 <- c(rep(NA, n_timesteps + 80), test_pred5, rep(NA,
nrow(vic_elec_test) - 80 - n_timesteps - n_forecast))


preds_ts <- vic_elec_test %>%
  select(Demand, Date) %>%
  add_column(
    ex_1 = test_pred1 * train_sd + train_mean,
    ex_2 = test_pred2 * train_sd + train_mean,
    ex_3 = test_pred3 * train_sd + train_mean,
    ex_4 = test_pred4 * train_sd + train_mean,
    ex_5 = test_pred5 * train_sd + train_mean) %>%
  pivot_longer(-Date) %>%
  update_tsibble(key = name)


preds_ts %>%
  autoplot() +
  scale_color_hue(h = c(80, 300), l = 70) +
  theme_minimal()
```




…