My starting point was one of the PyTorch implementations found on the net, namely, this one. If you search for “PyTorch convGRU” or “PyTorch convLSTM”, you will find stunning discrepancies in how these are realized – discrepancies not just in syntax and/or engineering ambition, but on the semantic level, right at the center of what the architectures may be expected to do. As they say, let the buyer beware. (Regarding the implementation I ended up porting, I am confident that while numerous optimizations will be possible, the basic mechanism matches my expectations.)

What do I expect? Let’s approach this task in a top-down way.

Input and output

The convLSTM’s input will be a time series of spatial data, each observation being of size `(time steps, channels, height, width)`.

Compare this with the usual RNN input format, be it in `torch` or Keras. In both frameworks, RNNs expect tensors of size `(timesteps, input_dim)`[1]. `input_dim` is \(1\) for univariate time series and greater than \(1\) for multivariate ones. Conceptually, we may match this to convLSTM’s `channels` dimension: There could be a single channel, for temperature, say – or there could be several, such as for pressure, temperature, and humidity. The two additional dimensions found in convLSTM, `height` and `width`, are spatial indexes into the data.

In sum, we want to be able to pass data that:

- consist of one or more features,
- evolve in time, and
- are indexed in two spatial dimensions.

How about the output? We want to be able to return forecasts for as many time steps as we have in the input sequence. This is something that `torch` RNNs do by default, while Keras equivalents do not. (You have to pass `return_sequences = TRUE` to obtain that effect.) If we’re interested in predictions for just a single point in time, we can always pick the last time step in the output tensor.

However, with RNNs, it is not all about outputs. RNN architectures also carry through hidden states.

What are hidden states? I carefully phrased that sentence to be as general as possible – deliberately circling around the confusion that, in my view, often arises at this point. We’ll attempt to clear up some of that confusion in a second, but let’s first finish our high-level requirements specification.

We want our convLSTM to be usable in different contexts and applications. Various architectures exist that make use of hidden states, most prominently perhaps, encoder-decoder architectures. Thus, we want our convLSTM to return those as well. Again, this is something a `torch` LSTM does by default, while in Keras it is achieved using `return_state = TRUE`.

Now though, it really is time for that interlude. We’ll sort out the ways things are called by both `torch` and Keras, and inspect what you get back from their respective GRUs and LSTMs.

Interlude: Outputs, states, hidden values … what’s what?

For this to remain an interlude, I summarize findings on a high level. The code snippets in the appendix show how to arrive at these results. Heavily commented, they probe return values from both Keras and `torch` GRUs and LSTMs. Running these will make the upcoming summaries seem a lot less abstract.

First, let's look at the ways you create an LSTM in both frameworks. (I will generally use LSTM as the "prototypical RNN example", and just mention GRUs when there are differences significant in the context in question.)

In Keras, to create an LSTM you may write something like this:

```
lstm <- layer_lstm(units = 1)
```

The `torch` equivalent would be:

```
lstm <- nn_lstm(
  input_size = 2, # number of input features
  hidden_size = 1 # number of hidden (and output!) features
)
```

Don't focus on `torch`'s `input_size` parameter for this discussion. (It's the number of features in the input tensor.) The parallel occurs between Keras' `units` and `torch`'s `hidden_size`. If you've been using Keras, you're probably thinking of `units` as the thing that determines output size (equivalently, the number of features in the output). So when `torch` lets us arrive at the same result using `hidden_size`, what does that mean? It means that somehow we're specifying the same thing, using different terminology. And it does make sense, since at every time step current input and previous hidden state are added[2]:

$$\mathbf{h}_t = \mathbf{W}_{x}\mathbf{x}_t + \mathbf{W}_{h}\mathbf{h}_{t-1}$$

Now, *about those hidden states*.

When a Keras LSTM is defined with `return_state = TRUE`, its return value is a structure of three entities called output, memory state, and carry state. In `torch`, the same entities are referred to as output, hidden state, and cell state. (In `torch`, we always get all of them.)

So are we dealing with three different types of entities? We are not.

The cell, or carry state is that special thing that sets apart LSTMs from GRUs deemed responsible for the "long" in "long short-term memory". Technically, it could be reported to the user at all points in time; as we'll see shortly though, it is not.

What about outputs and hidden, or memory states? Confusingly, these really are the same thing. Recall that for each input *item* in the input *sequence*, we're combining it with the previous state, resulting in a new state, to be made used of in the next step[3]:

$$\mathbf{h}_t = \mathbf{W}_{x}\mathbf{x}_t + \mathbf{W}_{h}\mathbf{h}_{t-1}$$

Now, say that we're interested in looking at just the final time step – that is, the default output of a Keras LSTM. From that point of view, we can consider those intermediate computations as "hidden". Seen like that, output and hidden states feel different.

However, we can also request to see the outputs for every time step. If we do so, there is no difference – the output**s** (plural) equal the hidden states. This can be verified using the code in the appendix.

Thus, of the three things returned by an LSTM, two are really the same. How about the GRU, then? As there is no “cell state”, we really have just one type of thing left over – call it outputs or hidden states.

Let’s summarize this in a table.

Table 1: RNN terminology. Comparing torch-speak and Keras-speak. In row 1, the terms are parameter names. In rows 2 and 3, they are pulled from current documentation.

Referring to this entity:	**`torch` says:**	**Keras says:**
Number of features in the output This determines both how many output features there are *and* the dimensionality of the hidden states.	`hidden_size`	`units`
Per-time-step output; latent state; intermediate state … This could be named “public state” in the sense that we, the users, are able to obtain *all values.*	hidden state	memory state
Cell state; inner state … (LSTM only) This could be named “private state” in that we are able to obtain a value *only for the last time step*. More on that in a second.	cell state	carry state

Now, about that public vs. private distinction. In both frameworks, we can obtain outputs (hidden states) for every time step. The cell state, however, we can access only for the very last time step. This is purely an implementation decision. As we’ll see when building our own recurrent module, there are no obstacles inherent in keeping track of cell states and passing them back to the user.

If you dislike the pragmatism of this distinction, you can always go with the math. When a new cell state has been computed (based on prior cell state, input, forget, and cell gates – the specifics of which we are not going to get into here), it is transformed to the hidden (a.k.a. output) state making use of yet another, namely, the output gate:

\[h_t = o_t \odot \tanh(c_t) \]

Definitely, then, hidden state (output, resp.) builds on cell state, adding additional modeling power.

Now it is time to get back to our original goal and build that convLSTM. First though, let’s summarize the return values obtainable from `torch` and Keras.

Table 2: Contrasting ways of obtaining various return values in `torch` vs. Keras. Cf. the appendix for complete examples.

To achieve this goal:	**in `torch` do:**	**in Keras do:**
access all intermediate outputs (= per-time-step outputs)	`ret[[1]]`	`return_sequences = TRUE`
access both “hidden state” (output) and “cell state” from final time step (only!)	`ret[[2]]`	`return_state = TRUE`
access all intermediate outputs and the final “cell state”	both of the above	`return_sequences = TRUE,` `return_state = TRUE`
access all intermediate outputs and “cell states” from all time steps	no way	no way

`convLSTM`, the plan

In both `torch` and Keras RNN architectures, single time steps are processed by corresponding `Cell` classes: There is an LSTM Cell matching the LSTM, a GRU Cell matching the GRU, and so on. We do the same for ConvLSTM. In `convlstm_cell()`, we first define what should happen to a single observation; then in `convlstm()`, we build up the recurrence logic.

Once we’re done, we create a dummy dataset, as reduced-to-the-essentials as can be. With more complex datasets, even artificial ones, chances are that if we don’t see any training progress, there are hundreds of possible explanations. We want a sanity check that, if failed, leaves no excuses. Realistic applications are left to future posts.

A single step: `convlstm_cell`

Our `convlstm_cell`’s constructor takes arguments `input_dim` , `hidden_dim`, and `bias`, just like a `torch` LSTM Cell.

But we’re processing two-dimensional input data. Instead of the usual affine combination of new input and previous state, we use a convolution of kernel size `kernel_size`. Inside `convlstm_cell`, it is `self$conv` that takes care of this.

Note how the `channels` dimension, which in the original input data would correspond to different variables, is creatively used to consolidate four convolutions into one: Each channel output will be passed to just one of the four cell gates. Once in possession of the convolution output, `forward()` applies the gate logic, resulting in the two types of states it needs to send back to the caller.

```
library(torch)
library(zeallot)

convlstm_cell <- nn_module(

  initialize = function(input_dim, hidden_dim, kernel_size, bias) {

    self$hidden_dim <- hidden_dim

    padding <- kernel_size %/% 2
    self$conv <- nn_conv2d(
      in_channels = input_dim + self$hidden_dim,
      # for each of input, forget, output, and cell gates
      out_channels = 4 * self$hidden_dim,
      kernel_size = kernel_size,
      padding = padding,
      bias = bias
    )
  },

  forward = function(x, prev_states) {

    c(h_prev, c_prev) %<-% prev_states

    combined <- torch_cat(list(x, h_prev), dim = 2)  # concatenate 
along channel axis
    combined_conv <- self$conv(combined)
    c(cc_i, cc_f, cc_o, cc_g) %<-% torch_split(combined_conv, 
self$hidden_dim, dim = 2)

    # input, forget, output, and cell gates (corresponding to torch's 
LSTM)
    i <- torch_sigmoid(cc_i)
    f <- torch_sigmoid(cc_f)
    o <- torch_sigmoid(cc_o)
    g <- torch_tanh(cc_g)

    # cell state
    c_next <- f * c_prev + i * g
    # hidden state
    h_next <- o * torch_tanh(c_next)

    list(h_next, c_next)
  },

  init_hidden = function(batch_size, height, width) {

    list(
      torch_zeros(batch_size, self$hidden_dim, height, width, device = 
self$conv$weight$device),
      torch_zeros(batch_size, self$hidden_dim, height, width, device = 
self$conv$weight$device))
  }
)
```

Now `convlstm_cell` has to be called for every time step. This is done by `convlstm`.

Iteration over time steps: `convlstm`

A `convlstm` may consist of several layers, just like a `torch` LSTM. For each layer, we are able to specify hidden and kernel sizes individually.

During initialization, each layer gets its own `convlstm_cell`. On call, `convlstm` executes two loops. The outer one iterates over layers. At the end of each iteration, we store the final pair `(hidden state, cell state)` for later reporting. The inner loop runs over input sequences, calling `convlstm_cell` at each time step.

We also keep track of intermediate outputs, so we'll be able to return the complete list of `hidden_state`s seen during the process. Unlike a `torch` LSTM, we do this *for every layer*.

```
convlstm <- nn_module(

  # hidden_dims and kernel_sizes are vectors, with one element for each
layer in n_layers
  initialize = function(input_dim, hidden_dims, kernel_sizes, n_layers,
bias = TRUE) {

    self$n_layers <- n_layers

    self$cell_list <- nn_module_list()

    for (i in 1:n_layers) {
      cur_input_dim <- if (i == 1) input_dim else hidden_dims[i - 1]
      self$cell_list$append(convlstm_cell(cur_input_dim,
hidden_dims[i], kernel_sizes[i], bias))
    }
  },

  # we always assume batch-first
  forward = function(x) {

    c(batch_size, seq_len, num_channels, height, width) %<-% x$size()

    # initialize hidden states
    init_hidden <- vector(mode = "list", length = self$n_layers)
    for (i in 1:self$n_layers) {
      init_hidden[[i]] <- self$cell_list[[i]]$init_hidden(batch_size,
height, width)
    }

    # list containing the outputs, of length seq_len, for each layer
    # this is the same as h, at each step in the sequence
    layer_output_list <- vector(mode = "list", length = self$n_layers)

    # list containing the last states (h, c) for each layer
    layer_state_list <- vector(mode = "list", length = self$n_layers)

    cur_layer_input <- x
    hidden_states <- init_hidden

    # loop over layers
    for (i in 1:self$n_layers) {

      # every layer's hidden state starts from 0 (non-stateful)
      c(h, c) %<-% hidden_states[[i]]
      # outputs, of length seq_len, for this layer
      # equivalently, list of h states for each time step
      output_sequence <- vector(mode = "list", length = seq_len)

      # loop over time steps
      for (t in 1:seq_len) {
        c(h, c) %<-% self$cell_list[[i]](cur_layer_input[ , t, , , ],
list(h, c))
        # keep track of output (h) for every time step
        # h has dim (batch_size, hidden_size, height, width)
        output_sequence[[t]] <- h
      }

      # stack hs for all time steps over seq_len dimension
      # stacked_outputs has dim (batch_size, seq_len, hidden_size,
height, width)
      # same as input to forward (x)
      stacked_outputs <- torch_stack(output_sequence, dim = 2)

      # pass the list of outputs (hs) to next layer
      cur_layer_input <- stacked_outputs

      # keep track of list of outputs or this layer
      layer_output_list[[i]] <- stacked_outputs
      # keep track of last state for this layer
      layer_state_list[[i]] <- list(h, c)
    }

    list(layer_output_list, layer_state_list)
  }

)
```

Calling the `convlstm`

Let's see the input format expected by `convlstm`, and how to access its different outputs.

Here is a suitable input tensor.

```
# batch_size, seq_len, channels, height, width
x <- torch_rand(c(2, 4, 3, 16, 16))
```

First we make use of a single layer.

```
model <- convlstm(input_dim = 3, hidden_dims = 5, kernel_sizes = 3,
n_layers = 1)

c(layer_outputs, layer_last_states) %<-% model(x)
```

We get back a list of length two, which we immediately split up into the two types of output returned: intermediate outputs from all layers, and final states (of both types) for the last layer.

With just a single layer, `layer_outputs[[1]]`holds all of the layer's intermediate outputs, stacked on dimension two.

```
dim(layer_outputs[[1]])
# [1]  2  4  5 16 16
```

`layer_last_states[[1]]`is a list of tensors, the first of which holds the single layer's final hidden state, and the second, its final cell state.

```
dim(layer_last_states[[1]][[1]])
# [1]  2  5 16 16
dim(layer_last_states[[1]][[2]])
# [1]  2  5 16 16
```

For comparison, this is how return values look for a multi-layer architecture.

```
model <- convlstm(input_dim = 3, hidden_dims = c(5, 5, 1), kernel_sizes
= rep(3, 3), n_layers = 3)
c(layer_outputs, layer_last_states) %<-% model(x)

# for each layer, tensor of size (batch_size, seq_len, hidden_size,
height, width)
dim(layer_outputs[[1]])
# 2  4  5 16 16
dim(layer_outputs[[3]])
# 2  4  1 16 16

# list of 2 tensors for each layer
str(layer_last_states)
# List of 3
#  $ :List of 2
#   ..$ :Float [1:2, 1:5, 1:16, 1:16]
#   ..$ :Float [1:2, 1:5, 1:16, 1:16]
#  $ :List of 2
#   ..$ :Float [1:2, 1:5, 1:16, 1:16]
#   ..$ :Float [1:2, 1:5, 1:16, 1:16]
#  $ :List of 2
#   ..$ :Float [1:2, 1:1, 1:16, 1:16]
#   ..$ :Float [1:2, 1:1, 1:16, 1:16]

# h, of size (batch_size, hidden_size, height, width)
dim(layer_last_states[[3]][[1]])
# 2  1 16 16

# c, of size (batch_size, hidden_size, height, width)
dim(layer_last_states[[3]][[2]])
# 2  1 16 16
```

Now we want to sanity-check this module with the simplest-possible dummy data.

Sanity-checking the `convlstm`

We generate black-and-white "movies" of diagonal beams successively translated in space.

Each sequence consists of six time steps, and each beam of six pixels. Just a single sequence is created manually. To create that one sequence, we start from a single beam:

```
library(torchvision)

beams <- vector(mode = "list", length = 6)
beam <- torch_eye(6) %>% nnf_pad(c(6, 12, 12, 6)) # left, right, top,
bottom
beams[[1]] <- beam
```

Using `torch_roll()` , we create a pattern where this beam moves up diagonally, and stack the individual tensors along the `timesteps` dimension.

```
for (i in 2:6) {
  beams[[i]] <- torch_roll(beam, c(-(i-1),i-1), c(1, 2))
}

init_sequence <- torch_stack(beams, dim = 1)
```

That's a single sequence. Thanks to `torchvision::transform_random_affine()`, we almost effortlessly produce a dataset of a hundred sequences. Moving beams start at random points in the spatial frame, but they all share that upward-diagonal motion.

```
sequences <- vector(mode = "list", length = 100)
sequences[[1]] <- init_sequence

for (i in 2:100) {
  sequences[[i]] <- transform_random_affine(init_sequence, degrees = 0,
translate = c(0.5, 0.5))
}

input <- torch_stack(sequences, dim = 1)

# add channels dimension
input <- input$unsqueeze(3)
dim(input)
# [1] 100   6  1  24  24
```

That's it for the raw data. Now we still need a `dataset` and a `dataloader`. Of the six time steps, we use the first five as input and try to predict the last one.

```
dummy_ds <- dataset(

  initialize = function(data) {
    self$data <- data
  },

  .getitem = function(i) {
    list(x = self$data[i, 1:5, ..], y = self$data[i, 6, ..])
  },

  .length = function() {
    nrow(self$data)
  }
)

ds <- dummy_ds(input)
dl <- dataloader(ds, batch_size = 100)
```

Here is a tiny-ish convLSTM, trained for motion prediction:

```
model <- convlstm(input_dim = 1, hidden_dims = c(64, 1), kernel_sizes =
c(3, 3), n_layers = 2)

optimizer <- optim_adam(model$parameters)

num_epochs <- 100

for (epoch in 1:num_epochs) {

  model$train()
  batch_losses <- c()

  for (b in enumerate(dl)) {

    optimizer$zero_grad()

    # last-time-step output from last layer
    preds <- model(b$x)[[2]][[2]][[1]]

    loss <- nnf_mse_loss(preds, b$y)
    batch_losses <- c(batch_losses, loss$item())

    loss$backward()
    optimizer$step()
  }

  if (epoch %% 10 == 0)
    cat(sprintf("\nEpoch %d, training loss:%3f\n", epoch,
mean(batch_losses)))
}
Epoch 10, training loss:0.008522

Epoch 20, training loss:0.008079

Epoch 30, training loss:0.006187

Epoch 40, training loss:0.003828

Epoch 50, training loss:0.002322

Epoch 60, training loss:0.001594

Epoch 70, training loss:0.001376

Epoch 80, training loss:0.001258
Epoch 90, training loss:0.001218

Epoch 100, training loss:0.001171
```

Loss decreases, but that in itself is not a guarantee the model has learned anything. Has it? Let's inspect its forecast for the very first sequence and see.

For printing, I'm zooming in on the relevant region in the 24x24-pixel frame. Here is the ground truth for time step six:

```
b$y[1, 1, 6:15, 10:19]
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  1  0  0  0  0  0  0  0
0  0  0  1  0  0  0  0  0  0
0  0  0  0  1  0  0  0  0  0
0  0  0  0  0  1  0  0  0  0
0  0  0  0  0  0  1  0  0  0
0  0  0  0  0  0  0  1  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
```

And here is the forecast. This does not look bad at all, given there was neither experimentation nor tuning involved.

```
round(as.matrix(preds[1, 1, 6:15, 10:19]), 2)
       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9] [,10]
 [1,]  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00     0
 [2,] -0.02  0.36  0.01  0.06  0.00  0.00  0.00  0.00  0.00     0
 [3,]  0.00 -0.01  0.71  0.01  0.06  0.00  0.00  0.00  0.00     0
 [4,] -0.01  0.04  0.00  0.75  0.01  0.06  0.00  0.00  0.00     0
 [5,]  0.00 -0.01 -0.01 -0.01  0.75  0.01  0.06  0.00  0.00     0
 [6,]  0.00  0.01  0.00 -0.07 -0.01  0.75  0.01  0.06  0.00     0
 [7,]  0.00  0.01 -0.01 -0.01 -0.07 -0.01  0.75  0.01  0.06     0
 [8,]  0.00  0.00  0.01  0.00  0.00 -0.01  0.00  0.71  0.00     0
 [9,]  0.00  0.00  0.00  0.01  0.01  0.00  0.03 -0.01  0.37     0
[10,]  0.00  0.00  0.00  0.00  0.00  0.00 -0.01 -0.01 -0.01     0
```

This should suffice for a sanity check. If you made it till the end, thanks for your patience! In the best case, you'll be able to apply this architecture (or a similar one) to your own data – but even if not, I hope you've enjoyed learning about `torch` model coding and/or RNN weirdness 😉

I, for one, am certainly looking forward to exploring convLSTMs on real-world problems in the near future. Thanks for reading!

Appendix

This appendix contains the code used to create tables 1 and 2 above.

Keras

LSTM

```
library(keras)

# batch of 3, with 4 time steps each and a single feature
input <- k_random_normal(shape = c(3L, 4L, 1L))
input

# default args
# return shape = (batch_size, units)
lstm <- layer_lstm(
  units = 1,
  kernel_initializer = initializer_constant(value = 1),
  recurrent_initializer = initializer_constant(value = 1)
)
lstm(input)

# return_sequences = TRUE
# return shape = (batch_size, time steps, units)
#
# note how for each item in the batch, the value for time step 4 equals
that obtained above
lstm <- layer_lstm(
  units = 1,
  return_sequences = TRUE,
  kernel_initializer = initializer_constant(value = 1),
  recurrent_initializer = initializer_constant(value = 1)
  # bias is by default initialized to 0
)
lstm(input)

# return_state = TRUE
# return shape = list of:
#                - outputs, of shape: (batch_size, units)
#                - "memory states" for the last time step, of shape:
(batch_size, units)
#                - "carry states" for the last time step, of shape:
(batch_size, units)
#
# note how the first and second list items are identical!
lstm <- layer_lstm(
  units = 1,
  return_state = TRUE,
  kernel_initializer = initializer_constant(value = 1),
  recurrent_initializer = initializer_constant(value = 1)
)
lstm(input)

# return_state = TRUE, return_sequences = TRUE
# return shape = list of:
#                - outputs, of shape: (batch_size, time steps, units)
#                - "memory" states for the last time step, of shape:
(batch_size, units)
#                - "carry states" for the last time step, of shape:
```

```
(batch_size, units)
#
# note how again, the "memory" state found in list item 2 matches the
final-time step outputs reported in item 1
lstm <- layer_lstm(
  units = 1,
  return_sequences = TRUE,
  return_state = TRUE,
  kernel_initializer = initializer_constant(value = 1),
  recurrent_initializer = initializer_constant(value = 1)
)
lstm(input)
```

GRU

```
# default args
# return shape = (batch_size, units)
gru <- layer_gru(
  units = 1,
  kernel_initializer = initializer_constant(value = 1),
  recurrent_initializer = initializer_constant(value = 1)
)
gru(input)

# return_sequences = TRUE
# return shape = (batch_size, time steps, units)
#
# note how for each item in the batch, the value for time step 4 equals
that obtained above
gru <- layer_gru(
  units = 1,
  return_sequences = TRUE,
  kernel_initializer = initializer_constant(value = 1),
  recurrent_initializer = initializer_constant(value = 1)
)
gru(input)

# return_state = TRUE
# return shape = list of:
#    - outputs, of shape: (batch_size, units)
#    - "memory" states for the last time step, of shape: (batch_size,
units)
#
# note how the list items are identical!
gru <- layer_gru(
  units = 1,
  return_state = TRUE,
  kernel_initializer = initializer_constant(value = 1),
  recurrent_initializer = initializer_constant(value = 1)
)
gru(input)
```

```
# return_state = TRUE, return_sequences = TRUE
# return shape = list of:
#    - outputs, of shape: (batch_size, time steps, units)
#    - "memory states" for the last time step, of shape: (batch_size,
units)
#
# note how again, the "memory state" found in list item 2 matches the
final-time-step outputs reported in item 1
gru <- layer_gru(
  units = 1,
  return_sequences = TRUE,
  return_state = TRUE,
  kernel_initializer = initializer_constant(value = 1),
  recurrent_initializer = initializer_constant(value = 1)
)
gru(input)
```

torch

LSTM (non-stacked architecture)

```
library(torch)

# batch of 3, with 4 time steps each and a single feature
# we will specify batch_first = TRUE when creating the LSTM
input <- torch_randn(c(3, 4, 1))
input

# default args
# return shape = (batch_size, units)
#
# note: there is an additional argument num_layers that we could use to
specify a stacked LSTM - effectively composing two LSTM modules
# default for num_layers is 1 though
lstm <- nn_lstm(
  input_size = 1, # number of input features
  hidden_size = 1, # number of hidden (and output!) features
  batch_first = TRUE # for easy comparability with Keras
)

nn_init_constant_(lstm$weight_ih_l1, 1)
nn_init_constant_(lstm$weight_hh_l1, 1)
nn_init_constant_(lstm$bias_ih_l1, 0)
nn_init_constant_(lstm$bias_hh_l1, 0)

# returns a list of length 2, namely
#   - outputs, of shape (batch_size, time steps, hidden_size) - given
we specified batch_first
#       Note 1: If this is a stacked LSTM, these are the outputs from
the last layer only.
#               For our current purpose, this is irrelevant, as we're
restricting ourselves to single-layer LSTMs.
```

```
#       Note 2: hidden_size here is equivalent to units in Keras - both
specify number of features
#  - list of:
#    - hidden state for the last time step, of shape (num_layers,
batch_size, hidden_size)
#    - cell state for the last time step, of shape (num_layers,
batch_size, hidden_size)
#      Note 3: For a single-layer LSTM, the hidden states are already
provided in the first list item.

lstm(input)
```

GRU (non-stacked architecture)

```
# default args
# return shape = (batch_size, units)
#
# note: there is an additional argument num_layers that we could use to
specify a stacked GRU - effectively composing two GRU modules
# default for num_layers is 1 though
gru <- nn_gru(
  input_size = 1, # number of input features
  hidden_size = 1, # number of hidden (and output!) features
  batch_first = TRUE # for easy comparability with Keras
)

nn_init_constant_(gru$weight_ih_l1, 1)
nn_init_constant_(gru$weight_hh_l1, 1)
nn_init_constant_(gru$bias_ih_l1, 0)
nn_init_constant_(gru$bias_hh_l1, 0)

# returns a list of length 2, namely
#   - outputs, of shape (batch_size, time steps, hidden_size) - given
we specified batch_first
#       Note 1: If this is a stacked GRU, these are the outputs from
the last layer only.
#               For our current purpose, this is irrelevant, as we're
restricting ourselves to single-layer GRUs.
#       Note 2: hidden_size here is equivalent to units in Keras - both
specify number of features
#  - list of:
#    - hidden state for the last time step, of shape (num_layers,
batch_size, hidden_size)
#    - cell state for the last time step, of shape (num_layers,
batch_size, hidden_size)
#       Note 3: For a single-layer GRU, these values are already
provided in the first list item.
gru(input)
```