

# El Niño, la Niña

ENSO refers to a changing pattern of sea surface temperatures and sea-level pressures occurring in the equatorial Pacific. From its three overall states, probably the best-known is El Niño. El Niño occurs when surface water temperatures in the eastern Pacific are higher than normal, and the strong winds that normally blow from east to west are unusually weak. The opposite conditions are termed La Niña. Everything in-between is classified as normal.

ENSO has great impact on the weather worldwide, and routinely harms ecosystems and societies through storms, droughts and flooding, possibly resulting in famines and economic crises. The best societies can do is try to adapt and mitigate severe consequences. Such efforts are aided by accurate forecasts, the further ahead the better.

Here, deep learning (DL) can potentially help: Variables like sea surface temperatures and pressures are given on a spatial grid – that of the earth – and as we know, DL is good at extracting spatial (e.g., image) features. For ENSO prediction, architectures like convolutional neural networks (Ham, Kim, and Luo (2019)) or convolutional-recurrent hybrids<sup>1</sup> are habitually used. One such hybrid is just our convLSTM; it operates on sequences of features given on a spatial grid. Today, thus, we'll be training a model for ENSO forecasting. This model will have a convLSTM for its central ingredient.

Before we start, a note. While our model fits well with architectures described in the relevant papers, the same cannot be said for amount of training data used. For reasons of practicality, we use actual observations only; consequently, we end up with a small (relative to the task) dataset. In contrast, research papers tend to make use of climate simulations<sup>2</sup>, resulting in significantly more data to work with.

From the outset, then, we don't expect stellar performance. Nevertheless, this should make for an interesting case study, and a useful code template for our readers to apply to their own data.

## Data

We will attempt to predict monthly average sea surface temperature in the Niño 3.4 region<sup>3</sup>, as represented by the Niño 3.4 Index, plus categorization as one of *El Niño*, *La Niña* or *neutral*<sup>4</sup>. Predictions will be based on prior monthly sea surface temperatures spanning a large portion of the globe.

On the input side, public and ready-to-use data may be downloaded from [Tokyo Climate Center](#); as to prediction targets, we obtain index and classification [here](#).

Input and target data both are provided monthly. They intersect in the time period ranging from 1891-01-01 to 2020-08-01; so this is the range of dates we'll be zooming in on.

## Input: Sea Surface Temperatures

Monthly sea surface temperatures are provided in a latitude-longitude grid of resolution 1°. Details of how the data were processed are available [here](#).

Data files are available in [GRIB](#) format; each file contains averages computed for a single month. We can either download individual files or [generate a text file of URLs](#) for download. In case you'd like to follow along with the post, you'll find the contents of the text file I generated in

the appendix. Once you've saved these URLs to a file, you can have R get the files for you like so:

```
purrr::walk(
  readLines("files"),
  function(f) download.file(url = f, destfile = basename(f))
)
```

From R, we can read GRIB files using [stars](#). For example:

```
# let's just quickly load all libraries we require to start with

library(torch)
library(tidyverse)
library(stars)
library(viridis)
library(ggthemes)

torch_manual_seed(777)

read_stars(file.path(grb_dir, "sst189101.grb"))
stars object with 2 dimensions and 1 attribute
attribute(s):
  sst189101.grb
Min.      :-274.9
1st Qu.   :-272.8
Median    :-259.1
Mean      :-260.0
3rd Qu.   :-248.4
Max.      :-242.8
NA's      :21001
dimension(s):
  from to offset delta          refsys point values
x    1 360      0      1 Coordinate System importe...  NA   NULL [x]
y    1 180     90     -1 Coordinate System importe...  NA   NULL [y]
```

So in this GRIB file, we have one attribute – which we know to be sea surface temperature – on a two-dimensional grid. As to the latter, we can complement what `stars` tells us with additional info found in the [documentation](#):

The east-west grid points run eastward from 0.5°E to 0.5°W, while the north-south grid points run northward from 89.5°S to 89.5°N.

We note a few things we'll want to do with this data. For one, the temperatures seem to be given in Kelvin, but with minus signs.<sup>5</sup> We'll remove the minus signs and convert to degrees Celsius for convenience. We'll also have to think about what to do with the `NA`s that appear for all non-maritime coordinates.

Before we get there though, we need to combine data from all files into a single data frame. This adds an additional dimension, time, ranging from 1891/01/01 to 2020/01/12:

```
grb <- read_stars(
  file.path(grb_dir, map(readLines("files", warn = FALSE), basename)),
  along = "time") %>%
```

```

st_set_dimensions(3,
                  values = seq(as.Date("1891-01-01"),
as.Date("2020-12-01"), by = "months"),
                  names = "time"
                  )

```

```

grb
stars object with 3 dimensions and 1 attribute
attribute(s), summary of first 1e+05 cells:

```

```

sst189101.grb
Min.      :-274.9
1st Qu.   :-273.3
Median    :-258.8
Mean      :-260.0
3rd Qu.   :-247.8
Max.      :-242.8
NA's      :33724
dimension(s):
      from   to offset delta      refsys point
values
x         1  360      0      1 Coordinate System importe...   NA
NULL [x]
y         1  180     90     -1 Coordinate System importe...   NA
NULL [y]
time      1 1560     NA     NA              Date      NA
1891-01-01,...,2020-12-01

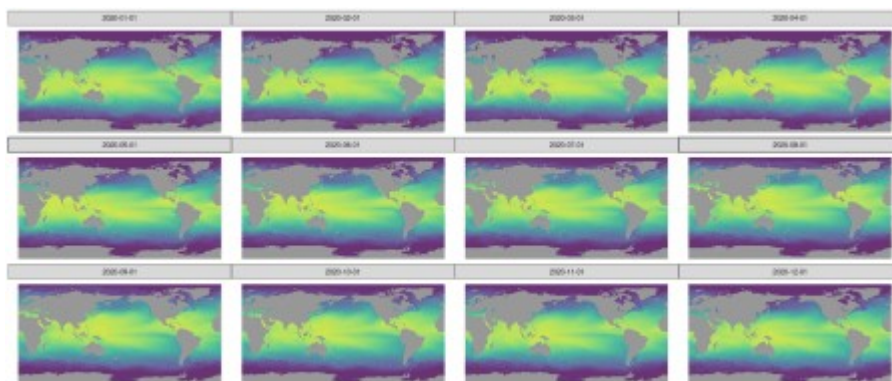
```

Let's visually inspect the spatial distribution of monthly temperatures for one year, 2020:

```

ggplot() +
  geom_stars(data = grb %>% filter(between(time, as.Date("2020-01-01"),
as.Date("2020-12-01"))), alpha = 0.8) +
  facet_wrap("time") +
  scale_fill_viridis() +
  coord_equal() +
  theme_map() +
  theme(legend.position = "none")

```



(#fig:unnamed-chunk-5)Monthly sea surface temperatures, 2020/01/01 – 2020/01/12.

## Target: Niño 3.4 Index

For the Niño 3.4 Index, we download the monthly [data](#) and, among the provided features, zoom in on two: the index itself (column `NINO34_MEAN`) and `PHASE`, which can be `E` (El Niño), `L` (La Niño) or `N` (neutral).

```
nino <- read_table2("ONI_NINO34_1854-2020.txt", skip = 9) %>%
  mutate(month = as.Date(paste0(YEAR, "-", `MON/MMM`, "-01"))) %>%
  select(month, NINO34_MEAN, PHASE) %>%
  filter(between(month, as.Date("1891-01-01"), as.Date("2020-08-01")))
  %>%
  mutate(phase_code = as.numeric(as.factor(PHASE)))

nrow(nino)
1556
```

Next, we look at how to get the data into a format convenient for training and prediction.

## Preprocessing

### Input

First, we remove all input data for points in time where ground truth data are still missing.

```
sst <- grb %>% filter(time <= as.Date("2020-08-01"))
```

Next, as is done by e.g. Ham, Kim, and Luo (2019), we only use grid points between 55° south and 60° north. This has the additional advantage of reducing memory requirements.

```
sst <- grb %>% filter(between(y, -55, 60))
```

```
dim(sst)
360, 115, 1560
```

As already alluded to, with the little data we have we can't expect much in terms of generalization. Still, we set aside a small portion of the data for validation, since we'd like for this post to serve as a useful template to be used with bigger datasets.

```
sst_train <- sst %>% filter(time < as.Date("1990-01-01"))
sst_valid <- sst %>% filter(time >= as.Date("1990-01-01"))
```

From here on, we work with R arrays.

```
sst_train <- as.tbl_cube.stars(sst_train)$mets[[1]]
sst_valid <- as.tbl_cube.stars(sst_valid)$mets[[1]]
```

Conversion to degrees Celsius is not strictly necessary, as initial experiments showed a slight performance increase due to normalizing the input, and we're going to do that anyway. Still, it reads nicer to humans than Kelvin.

```
sst_train <- sst_train + 273.15
quantile(sst_train, na.rm = TRUE)
      0%      25%      50%      75%     100%
-1.8000 12.9975 21.8775 26.8200 34.3700
```

Not at all surprisingly, global warming is evident from inspecting temperature distribution on the validation set (which was chosen to span the last thirty-one years).

```
sst_valid <- sst_valid + 273.15
quantile(sst_valid, na.rm = TRUE)
      0%      25%      50%      75%     100%
-1.800 13.425 22.335 27.240 34.870
```

The next-to-last step normalizes both sets according to training mean and variance.

```
train_mean <- mean(sst_train, na.rm = TRUE)
train_sd <- sd(sst_train, na.rm = TRUE)

sst_train <- (sst_train - train_mean) / train_sd

sst_valid <- (sst_valid - train_mean) / train_sd
```

Finally, what should we do about the NA entries? We set them to zero, the (training set) mean. That may not be enough of an action though: It means we're feeding the network roughly 30% misleading data. This is something we're not done with yet.

```
sst_train[is.na(sst_train)] <- 0
sst_valid[is.na(sst_valid)] <- 0
```

## Target

The target data are split analogously. Let's check though: Are phases (categorizations) distributedly similarly in both sets?

```
nino_train <- nino %>% filter(month < as.Date("1990-01-01"))
nino_valid <- nino %>% filter(month >= as.Date("1990-01-01"))

nino_train %>% group_by(phase_code, PHASE) %>% summarise(count = n(),
  avg = mean(NINO34_MEAN))
# A tibble: 3 x 4
# Groups:   phase_code [3]
  phase_code PHASE count    avg
1          1 E      301  27.7
2          2 L      333  25.6
3          3 N      554  26.7

nino_valid %>% group_by(phase_code, PHASE) %>% summarise(count = n(),
  avg = mean(NINO34_MEAN))
# A tibble: 3 x 4
# Groups:   phase_code [3]
  phase_code PHASE count    avg
1          1 E       93  28.1
2          2 L       93  25.9
3          3 N      182  27.2
```

This doesn't look too bad. Of course, we again see the overall rise in temperature, irrespective of phase.

Lastly, we normalize the index, same as we did for the input data.

```
train_mean_nino <- mean(nino_train$NINO34_MEAN)
```

```
train_sd_nino <- sd(nino_train$NINO34_MEAN)
```

```
nino_train <- nino_train %>% mutate(NINO34_MEAN = scale(NINO34_MEAN,  
center = train_mean_nino, scale = train_sd_nino))  
nino_valid <- nino_valid %>% mutate(NINO34_MEAN = scale(NINO34_MEAN,  
center = train_mean_nino, scale = train_sd_nino))
```

On to the `torch` dataset.

## Torch dataset

The dataset is responsible for correctly matching up inputs and targets.

Our goal is to take six months of global sea surface temperatures and predict the Niño 3.4 Index for the following month. Input-wise, the model will expect the following format semantics:

`batch_size * timesteps * width * height * channels`, where

- `batch_size` is the number of observations worked on in one round of computations,
- `timesteps` chains consecutive observations from adjacent months,
- `width` and `height` together constitute the spatial grid, and
- `channels` corresponds to available visual channels in the “image”.

In `.getitem()`, we select the consecutive observations, starting at a given index, and stack them in dimension one. (One, not two, as batches will only start to exist once the `dataloader` comes into play.)

Now, what about the target? Our ultimate goal was – is – predicting the Niño 3.4 Index. However, as you see we define three targets: One is the index, as expected; an additional one holds the spatially-gridded sea surface temperatures for the prediction month. Why? Our main instrument, the most prominent constituent of the model, will be a `convLSTM`, an architecture designed for *spatial* prediction. Thus, to train it efficiently, we want to give it the opportunity to predict values on a spatial grid. So far so good; but there’s one more target, the phase/category. This was added for experimentation purposes: Maybe predicting *both* index *and* phase helps in training?

Finally, here is the code for the dataset. In our experiments, we based predictions on inputs from the preceding six months (`n_timesteps <- 6`). This is a parameter you might want to play with, though.

```
n_timesteps <- 6
```

```
enso_dataset <- dataset(  
  name = "enso_dataset",
```

```
  initialize = function(sst, nino, n_timesteps) {  
    self$sst <- sst  
    self$nino <- nino  
    self$n_timesteps <- n_timesteps  
  },
```

```

    .getitem = function(i) {
      x <- torch_tensor(self$sst[, , i:(n_timesteps + i - 1)]) # (360,
115, n_timesteps)
      x <- x$permute(c(3,1,2))$unsqueeze(2) # (n_timesteps, 1, 360, 115)

      y1 <- torch_tensor(self$sst[, , n_timesteps + i])$unsqueeze(1) #
(1, 360, 115)
      y2 <- torch_tensor(self$nino$NINO34_MEAN[n_timesteps + i])
      y3 <- torch_tensor(self$nino$phase_code[n_timesteps +
i])$squeeze()$to(torch_long())
      list(x = x, y1 = y1, y2 = y2, y3 = y3)
    },

    .length = function() {
      nrow(self$nino) - n_timesteps
    }
  )

valid_ds <- enso_dataset(sst_valid, nino_valid, n_timesteps)

```

## Dataloaders

After the custom dataset, we create the – pretty typical – dataloaders, making use of a batch size of 4.

```

batch_size <- 4

train_dl <- train_ds %>% dataloader(batch_size = batch_size, shuffle =
TRUE)

valid_dl <- valid_ds %>% dataloader(batch_size = batch_size)

```

Next, we proceed to model creation.

## Model

The model's main ingredient is the convLSTM introduced in a [prior post](#). For convenience, we reproduce the code in the appendix.

Besides the convLSTM, the model makes use of three convolutional layers, a batchnorm layer and five linear layers. The logic is the following.

First, the convLSTM job is to predict the next month's sea surface temperatures on the spatial grid. For that, we *almost* just return its final state, - almost: We use `self$conv1` to reduce the number channels to one.

For predicting index and phase, we then need to flatten the grid, as we require a single value each. This is where the additional conv layers come in. We *do* hope they'll aid in learning, but we also want to reduce the number of parameters a bit, downsizing the grid (`strides = 2` and `strides = 3`, resp.) a bit before the upcoming `torch_flatten()`.

Once we have a flat structure, learning is shared between the tasks of index and phase

prediction (self\$linear), until finally their paths split (self\$cont and self\$cat, resp.), and they return their separate outputs.

(The batchnorm? I'll comment on that in the [Discussion](#).)

```
model <- nn_module(

  initialize = function(channels_in,
                        convlstm_hidden,
                        convlstm_kernel,
                        convlstm_layers) {

    self$n_layers <- convlstm_layers

    self$convlstm <- convlstm(
      input_dim = channels_in,
      hidden_dims = convlstm_hidden,
      kernel_sizes = convlstm_kernel,
      n_layers = convlstm_layers
    )

    self$conv1 <-
      nn_conv2d(
        in_channels = 32,
        out_channels = 1,
        kernel_size = 5,
        padding = 2
      )

    self$conv2 <-
      nn_conv2d(
        in_channels = 32,
        out_channels = 32,
        kernel_size = 5,
        stride = 2
      )

    self$conv3 <-
      nn_conv2d(
        in_channels = 32,
        out_channels = 32,
        kernel_size = 5,
        stride = 3
      )

    self$linear <- nn_linear(33408, 64)

    self$b1 <- nn_batch_norm1d(num_features = 64)

    self$cont <- nn_linear(64, 128)
    self$cat <- nn_linear(64, 128)
```



```

        self$cont_output <- nn_linear(128, 1)
        self$cat_output <- nn_linear(128, 3)

    },

    forward = function(x) {

        ret <- self$convlstm(x)
        layer_last_states <- ret[[2]]
        last_hidden <- layer_last_states[[self$n_layers]][[1]]

        next_sst <- last_hidden %>% self$conv1()

        c2 <- last_hidden %>% self$conv2()
        c3 <- c2 %>% self$conv3()

        flat <- torch_flatten(c3, start_dim = 2)
        common <- self$linear(flat) %>% self$b3() %>% nnf_relu()

        next_temp <- common %>% self$cont() %>% nnf_relu() %>%
self$cont_output()
        next_nino <- common %>% self$cat() %>% nnf_relu() %>%
self$cat_output()

        list(next_sst, next_temp, next_nino)

    }

)

```

Next, we instantiate a pretty small-ish model. You're more than welcome to experiment with larger models, but training time as well as GPU memory requirements will increase.

```

net <- model(
  channels_in = 1,
  convlstm_hidden = c(16, 16, 32),
  convlstm_kernel = c(3, 3, 5),
  convlstm_layers = 3
)

device <- torch_device(if (cuda_is_available()) "cuda" else "cpu")

net <- net$to(device = device)
net
An `nn_module` containing 2,389,605 parameters.

```

---

#### — Modules —

- convlstm: #182,080 parameters
- conv1: #801 parameters
- conv2: #25,632 parameters
- conv3: #25,632 parameters

- linear: #2,138,176 parameters
- b1: #128 parameters
- cont: #8,320 parameters
- cat: #8,320 parameters
- cont\_output: #129 parameters
- cat\_output: #387 parameters

## Training

We have three model outputs. How should we combine the losses?

Given that the main goal is predicting the index, and the other two outputs are essentially means to an end, I found the following combination rather effective:

```
# weight for sea surface temperature prediction
lw_sst <- 0.2

# weight for prediction of El Nino 3.4 Index
lw_temp <- 0.4

# weight for phase prediction
lw_nino <- 0.4
```

The training process follows the pattern seen in all `torch` posts so far: For each epoch, loop over the training set, backpropagate, check performance on validation set.

*But*, when we did the pre-processing, we were aware of an imminent problem: the missing temperatures for continental areas, which we set to zero. As a sole measure, this approach is clearly insufficient. What if we had chosen to use latitude-dependent averages? Or interpolation? Both may be better than a global average, but both have their problems as well. Let's at least alleviate negative consequences by not using the respective pixels for spatial loss calculation. This is taken care of by the following line below:

```
sst_loss <- nnf_mse_loss(sst_output[sst_target != 0],
sst_target[sst_target != 0])
```

Here, then, is the complete training code.

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

num_epochs <- 50

train_batch <- function(b) {

  optimizer$zero_grad()
  output <- net(b$x$to(device = device))

  sst_output <- output[[1]]
  sst_target <- b$y1$to(device = device)

  sst_loss <- nnf_mse_loss(sst_output[sst_target != 0],
sst_target[sst_target != 0])
  temp_loss <- nnf_mse_loss(output[[2]], b$y2$to(device = device))
  nino_loss <- nnf_cross_entropy(output[[3]], b$y3$to(device = device))
```

```

    loss <- lw_sst * sst_loss + lw_temp * temp_loss + lw_nino * nino_loss
    loss$backward()
    optimizer$step()

    list(sst_loss$item(), temp_loss$item(), nino_loss$item(),
loss$item())

}

valid_batch <- function(b) {

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

    sst_output <- output[[1]]
    sst_target <- b$y1$to(device = device)

    sst_loss <- nnf_mse_loss(sst_output[sst_target != 0],
sst_target[sst_target != 0])
    temp_loss <- nnf_mse_loss(output[[2]], b$y2$to(device = device))
    nino_loss <- nnf_cross_entropy(output[[3]], b$y3$to(device = device))

    loss <-
        lw_sst * sst_loss + lw_temp * temp_loss + lw_nino * nino_loss

    list(sst_loss$item(),
        temp_loss$item(),
        nino_loss$item(),
        loss$item())
}

for (epoch in 1:num_epochs) {

    net$train()

    train_loss_sst <- c()
    train_loss_temp <- c()
    train_loss_nino <- c()
    train_loss <- c()

    coro::loop(for (b in train_dl) {
        losses <- train_batch(b)
        train_loss_sst <- c(train_loss_sst, losses[[1]])
        train_loss_temp <- c(train_loss_temp, losses[[2]])
        train_loss_nino <- c(train_loss_nino, losses[[3]])
        train_loss <- c(train_loss, losses[[4]])
    })

    cat(
        sprintf(
            "\nEpoch %d, training: loss: %3.3f sst: %3.3f temp: %3.3f nino:
%3.3f \n",

```

```

        epoch, mean(train_loss), mean(train_loss_sst),
mean(train_loss_temp), mean(train_loss_nino)
    )
)

net$eval()

valid_loss_sst <- c()
valid_loss_temp <- c()
valid_loss_nino <- c()
valid_loss <- c()

coro::loop(for (b in valid_dl) {
    losses <- valid_batch(b)
    valid_loss_sst <- c(valid_loss_sst, losses[[1]])
    valid_loss_temp <- c(valid_loss_temp, losses[[2]])
    valid_loss_nino <- c(valid_loss_nino, losses[[3]])
    valid_loss <- c(valid_loss, losses[[4]])

})

cat(
    sprintf(
        "\nEpoch %d, validation: loss: %3.3f sst: %3.3f temp: %3.3f nino:
%3.3f \n",
        epoch, mean(valid_loss), mean(valid_loss_sst),
mean(valid_loss_temp), mean(valid_loss_nino)
    )
)

torch_save(net, paste0(
    "model_", epoch, "_", round(mean(train_loss), 3), "_",
round(mean(valid_loss), 3), ".pt"
))

}

```

When I ran this, performance on the training set decreased in a not-too-fast, but continuous way, while validation set performance kept fluctuating. For reference, total (composite) losses looked like this:

Epoch	Training	Validation
10	0.336	0.633
20	0.233	0.295
30	0.135	0.461
40	0.099	0.903
50	0.061	0.727

Thinking of the size of the validation set - thirty-one years, or equivalently, 372 data points – those fluctuations may not be all too surprising.

## Predictions

Now losses tend to be abstract; let's see what actually gets predicted. We obtain predictions for index values and phases like so ...

```
net$eval()

pred_index <- c()
pred_phase <- c()

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

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

  pred_index <- c(pred_index, output[[2]]$to(device = "cpu"))
  pred_phase <- rbind(pred_phase, as.matrix(output[[3]]$to(device =
"cpu")))

})
```

... and combine these with the ground truth, stripping off the first six rows (six was the number of timesteps used as predictors):

```
valid_perf <- data.frame(
  actual_temp = nino_valid$NINO34_MEAN[(batch_size +
1):nrow(nino_valid)] * train_sd_nino + train_mean_nino,
  actual_nino = factor(nino_valid$phase_code[(batch_size +
1):nrow(nino_valid)]),
  pred_temp = pred_index * train_sd_nino + train_mean_nino,
  pred_nino = factor(pred_phase %>% apply(1, which.max))
)
```

For the phase, we can generate a confusion matrix:

```
yardstick::conf_mat(valid_perf, actual_nino, pred_nino)
```

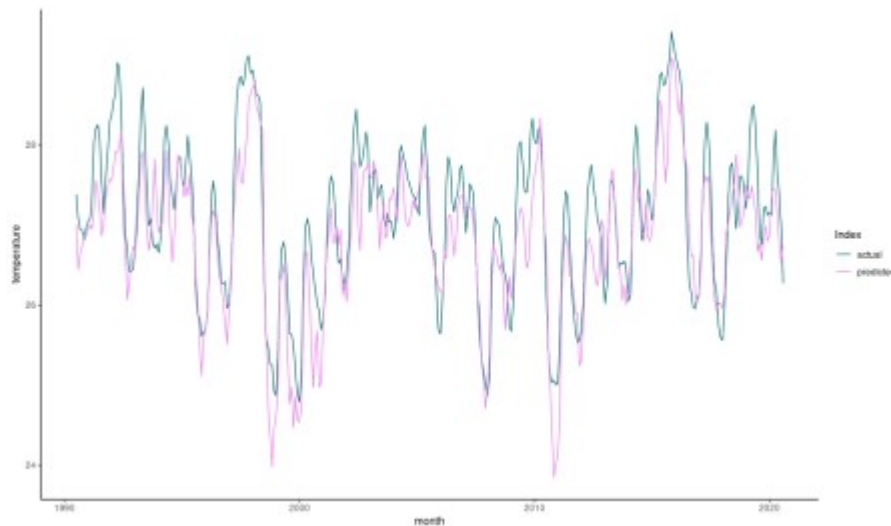
	Truth		
Prediction	1	2	3
1	70	0	43
2	0	47	10
3	23	46	123

This looks better than expected (based on the losses). Phases 1 and 2 correspond to El Niño and La Niña, respectively, and these get sharply separated.

What about the Niño 3.4 Index? Let's plot predictions versus ground truth:

```
valid_perf <- valid_perf %>%
  select(actual = actual_temp, predicted = pred_temp) %>%
  add_column(month = seq(as.Date("1990-07-01"), as.Date("2020-08-01"),
by = "months")) %>%
  pivot_longer(-month, names_to = "Index", values_to = "temperature")

ggplot(valid_perf, aes(x = month, y = temperature, color = Index)) +
  geom_line() +
  scale_color_manual(values = c("#006D6F", "#B2FFFF")) +
  theme_classic()
```



(#fig:unnamed-chunk-28)Nino 3.4 Index: Ground truth vs. predictions (validation set).

This does not look bad either. However, we need to keep in mind that we're predicting just a single time step ahead. We probably should not overestimate the results. Which leads directly to the discussion.

## Discussion

When working with small amounts of data, a lot can be learned by quick-ish experimentation. However, when *at the same time*, the task is complex, one should be cautious extrapolating.

For example, well-established regularizers such as batchnorm and dropout, while intended to improve generalization to the validation set, may turn out to severely impede training itself. This is the story behind the one batchnorm layer I kept (I did try having more), and it is also why there is no dropout.

One lesson to learn from this experience then is: Make sure the amount of data matches the complexity of the task. This is what we see in the ENSO prediction papers published on arxiv.

If we should treat the results with caution, why even publish the post?

For one, it shows an application of convLSTM to real-world data, employing a reasonably complex architecture and illustrating techniques like custom losses and loss masking. Similar architectures and strategies should be applicable to a wide range of real-world tasks – basically, whenever predictors in a time-series problem are given on a spatial grid.

Secondly, the application itself – forecasting an atmospheric phenomenon that greatly affects ecosystems as well as human well-being – seems like an excellent use of deep learning. Applications like these stand out as all the more worthwhile as the same cannot be said of everything deep learning is – and will be, barring effective regulation – used for.

Thanks for reading!

## Appendix

### A1: List of GRB files

To be put into a text file for use with `purrr::walk( ... download.file ... )`.

[https://ds.data.jma.go.jp/tcc/tcc/products/el\\_nino/cobesst/gpvddata/1891-1899/sst189101.grb](https://ds.data.jma.go.jp/tcc/tcc/products/el_nino/cobesst/gpvddata/1891-1899/sst189101.grb)

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2010-2019/sst201902.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2010-2019/sst201903.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2010-2019/sst201904.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2010-2019/sst201905.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2010-2019/sst201906.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2010-2019/sst201907.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2010-2019/sst201908.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2010-2019/sst201909.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2010-2019/sst201910.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2010-2019/sst201911.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2010-2019/sst201912.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2020-2029/sst202001.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2020-2029/sst202002.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2020-2029/sst202003.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2020-2029/sst202004.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2020-2029/sst202005.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2020-2029/sst202006.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2020-2029/sst202007.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2020-2029/sst202008.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2020-2029/sst202009.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2020-2029/sst202010.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2020-2029/sst202011.grb>  
<https://ds.data.jma.go.jp/tcc/tcc/products/elnino/cobesst/gpvdata/2020-2029/sst202012.grb>

## A2: convlstm code

For an in-depth explanation of `convlstm`, see [the blog post](#).



```

library(torch)
library(torchvision)

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) {

    h_prev <- prev_states[[1]]
    c_prev <- prev_states[[2]]

    combined <- torch_cat(list(x, h_prev), dim = 2) # concatenate
    along channel axis
    combined_conv <- self$conv(combined)
    gate_convs <- torch_split(combined_conv, self$hidden_dim, dim = 2)
    cc_i <- gate_convs[[1]]
    cc_f <- gate_convs[[2]]
    cc_o <- gate_convs[[3]]
    cc_g <- gate_convs[[4]]

    # 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))
  }
)

```

```

convlstm <- nn_module(

```

```

  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) {

```

```

    batch_size <- x$size()[1]

```

```

    seq_len <- x$size()[2]

```

```

    height <- x$size()[4]

```

```

    width <- x$size()[5]

```

```

    # 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)

```

```

h_c <- hidden_states[[i]]
h <- h_c[[1]]
c <- h_c[[2]]
# 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 timesteps
for (t in 1:seq_len) {
  h_c <- self$cell_list[[i]](cur_layer_input[ , t, , ], list(h,
c))
  h <- h_c[[1]]
  c <- h_c[[2]]
  # keep track of output (h) for every timestep
  # h has dim (batch_size, hidden_size, height, width)
  output_sequence[[t]] <- h
}

# stack hs for all timesteps 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 of 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)

}

)

```