Many [news reports](https://www.businessmodelsinc.com/machines/) scare us with machines taking over our jobs in the not too distant future. Common examples of take-over targets include professions like truck drivers, lawyers and accountants. In this article we will explore how far machines are from replacing **us** (R programmers) in writing Shiny code. **Spoiler alert**: you should not be worried about your obsolescence right now. You will see in a minute that we’re not quite there yet. I’m just hoping to show you in an entertaining way some easy applications of a simple model of a recurrent neural network implemented in an [R version](https://keras.rstudio.com/) of [Keras](https://keras.io/)

Let’s formulate our problem once again precisely: ***we want to generate Shiny code character by character with a neural network***.

**Background**

To achieve that we would need a **recurrent neural network** (RNN). By definition such a network does a pretty good job with **time series**. Right now you might be asking yourself, *what*?  We defined our problem as a text mining issue; where is temporal dependency here?! Well, imagine a programmer typing characters on his/her keyboard, one by one, every **time step**. It would also be nice if our network captured long-range dependencies such as, for instance, a curly bracket in the 1021st line of code that can refer to a “for” loop from  line 352 (that would be a long loop though). Fortunately, RNNs are perfect for that because they can (in theory) memorize the influence of a signal from the distant past to a present data sample.

I will not get into details on how recurrent neural networks work here, as I believe that there are a lot of fantastic resources [online elsewhere.](https://colah.github.io/posts/2015-08-Understanding-LSTMs/) Let me just briefly mention that some of the regular recurrent networks suffer from a *vanishing gradient* problem. As a result, networks with such architectures are notoriously difficult to train. That’s why machine learning researchers started looking for more robust solutions. These are provided by a gating mechanism that helps to teach a network long-term dependencies.

The first such solution was introduced in 1997 as a **Long Short Term Memory** neuron (LSTM). It consists of three gates: *input*, *forget* and *output,* that together prevent the gradient from vanishing in further time steps. A simplified version of LSTM that still achieves good performance is the **Gated Recurrent Unit** (GRU) introduced in 2014. In this solution, *forget* and *input* gates are merged into one *update* gate. In our implementation we will use a layer of GRU units.

Most of my code relies on an excellent example from Chapter 8 in [Deep Learning with R](https://www.manning.com/books/deep-learning-with-r) by François Chollet. I recommend this book wholeheartedly to everyone interested in practical basics of neural networks. Since I think that François can explain to you his implementation better than I could , I’ll just leave you with it and get to the part I modified or added.

**Experiment**

Before we get to the model, we need some training data. As we don’t want to generate just  any code, but specifically Shiny code , we need to find enough training samples. For that, I scraped the data mainly from this official [shiny examples repository](https://github.com/rstudio/shiny-examples/) and added some of our [semantic examples](https://github.com/Appsilon/shiny.semantic/tree/develop/examples). As a result I generated 1300 lines of Shiny code.

Second, I played with  several network architectures and looked for a balance between speed of training, accuracy and model complexity. After some experiments, I found a suitable network for our purposes:

model <- keras\_model\_sequential() %>%

layer\_gru(units = 100, input\_shape = c(maxlen, length(chars))) %>%

layer\_dense(units = length(chars), activation = "softmax")

(BTW If you want to find out more about Keras in R, I invite you to take a look at a nice introduction by [Michał](https://appsilon.com/ship-recognition-in-satellite-imagery-part-i/)).

I trained the above model for 50 epochs with a learning rate of 0.02. I experimented with different values of a temperature parameter too. Temperature is used to control the randomness of a prediction by scaling the logits (output of a last layer) before applying the softmax function. To illustrate, let’s  have a look at the output of the network predictions with temperature = 0.07.

nput$n)

})

# gen

dorre out: t", " ras <- ss)

}, # l imat")

})

}

# tageerl itht oimang = shndabres(h4t 1")

})

sses$ypabs viog hthewest onputputpung w do panetatstaserval = 1alin hs <----- geo verdpasyex(")

})

send tmonammm(asera d ary vall wa g xb =1iomm(dat\_ngg( ----dater(

# fu t coo ------ 1ang aoplono----i\_dur d"),

o tehing 1 ch mout = cor;")o}) <- t <- coan t

d i

and with temperature = 1:

filectinput <- ren({

# goith an htmm the oblsctr th verichend dile distr(input$dateretcaption$print\_om <- ren({

th cond filen(io outputs

# the ion tppet chooww.h vichecheckboartcarp" = show(dy),

simptect = select)

})

})

# funutput$datetable <- ren({

heag(heig= x(input$obr))

})

)

)

function({

suiphed = simplenter = "opter")

)

)

)

I think that both examples are already quite impressive, given the limited training data we had. In the first case, the network is more confident about its choices but also quite prone to repetitions (many spaces follow spaces, letters follow letters and so on). The latter, from a long, loooong distance looks way closer to Shiny code. Obviously, it’s still gibberish, but look! There is a nice function call heag(heig= x(input$obr)), object property input$obr, comment # goith and even variable assignment filectinput <- ren({. Isn’t that cool?

Let’s have a look now at the evolution of training after 5 epochs:

finp <- r ctived = "text",

"dachintion <- pepristexplet({

ut <- fendertext({

ftable('checkbs cutpanel changlis input dowcter selecter base bar ----

10 epochs:

# data

<- pasht(brch(null)

] ),

# input$c a couten quift to col c( expctfrlren beteracing changatput: pp--))

})

20 epochs:

fine asc i) {

foutput("apputc"in"),

text(teat(input$contrs)

# th render a number butt summaryerver pation ion tre

# chapte the gendate ion. bhthect.hate whtn hblo

As you can see, after each training the generated text becomes increasingly structured.

**Final Thoughts**

I appreciate that some of you might not be as impressed as I was. Frankly speaking, I almost hear all of these Shiny programmers saying: “Phew… my job is secure then!” Yeah, yeah, sure it is… For now! Remember that these models will probably improve over time. I  challenge you to play with different architectures and train some better models based on this example.

And for completeness, here’s the code I used to generate the fake Shiny code above:

library(keras)

library(stringr)

path <- "shinyappstextdata.dat" # input data path

text <- tolower(readChar(path, [file.info](http://file.info)(path)$size)) # loading data

# ------------------ Data preprocessing

maxlen <- 20

step <- 3

text\_indexes <- seq(1, nchar(text) - maxlen, by = step)

sentences <- str\_sub(text, text\_indexes, text\_indexes + maxlen - 1)

next\_chars <- str\_sub(text, text\_indexes + maxlen, text\_indexes + maxlen)

cat("Number of sequences: ", length(sentences), "\n")

chars <- unique(sort(strsplit(text, "")[[1]]))

cat("Unique characters:", length(chars), "\n")

char\_indices <- 1:length(chars)

names(char\_indices) <- chars

cat("Vectorization...\n")

x <- array(0L, dim = c(length(sentences), maxlen, length(chars)))

y <- array(0L, dim = c(length(sentences), length(chars)))

for (i in 1:length(sentences)) {

sentence <- strsplit(sentences[[i]], "")[[1]]

for (t in 1:length(sentence)) {

char <- sentence[[t]]

x[i, t, char\_indices[[char]]] <- 1

}

next\_char <- next\_chars[[i]]

y[i, char\_indices[[next\_char]]] <- 1

}

# ------------------ RNN model training

model <- keras\_model\_sequential() %>%

layer\_gru(units = 100, input\_shape = c(maxlen, length(chars))) %>%

layer\_dense(units = length(chars), activation = "softmax")

optimizer <- optimizer\_rmsprop(lr = 0.02) model %>% compile(

loss = "categorical\_crossentropy",

optimizer = optimizer

)

model %>% fit(x, y, batch\_size = 128, epochs = 50)

# ------------------ Predictions evaluation

sample\_next\_char <- function(preds, temperature = 1.0) {

preds <- as.numeric(preds)

preds <- log(preds) / temperature

exp\_preds <- exp(preds)

preds <- exp\_preds / sum(exp\_preds)

which.max(t(rmultinom(1, 1, preds)))

}

nr\_of\_character\_to\_generate <- 500

start\_index <- sample(1:(nchar(text) - maxlen - 1), 1)

seed\_text <- str\_sub(text, start\_index, start\_index + maxlen - 1)

temperature <- 1.0

cat(seed\_text, "\n")

generated\_text <- seed\_text

for (i in 1:nr\_of\_character\_to\_generate) {

sampled <- array(0, dim = c(1, maxlen, length(chars)))

generated\_chars <- strsplit(generated\_text, "")[[1]]

for (t in 1:length(generated\_chars)) {

char <- generated\_chars[[t]]

sampled[1, t, char\_indices[[char]]] <- 1

}

preds <- model %>% predict(sampled, verbose = 0)

next\_index <- sample\_next\_char(preds[1,], temperature)

next\_char <- chars[[next\_index]]

generated\_text <- paste0(generated\_text, next\_char)

generated\_text <- substring(generated\_text, 2)

cat(next\_char)

}