New York University: Machine Learning in Economics

Lecture 11: Neural Networks for Time Series

Dr. George Lentzas

Introduction

What are Recurrent Neural Networks?

- A major characteristic of most neural nets is that they posses no memory. Each input shown to them is processed independently and there is no state of the worlds that is kept in memory between inputs. Equivalently. you can shuffle the order of the inputs without alterning the neural network.
- In order to process time series with such a model you have to show the entire sequence to the network at once, effectively turning it to a signle data point. SUch neural networks are called *feedforwards networks*.
- A recurrent network instead processes sequences by iterating through the sequence elements and maintaining a state containing information about what the model has seen so far. A key characteristic of recurrent networks is the state function which summarizes what the state of the world created using information from time t-1 .

Introduction

What are Recurrent Neural Networks?

- So Recurrent neural networks (RNNs) are a family of neural networks for analyzing sequential data.
- Analyzing sequential data requires some extra assumptions. Consider the model

$$h_t = g_t\left(x_t, x_{t-1}, \ldots, x_1\right)$$

- This function g_t takes as input the entire sequence of past observations and is therefore a function that changes over time. This would make learning it a very difficult, if not impossible task.

Introduction

What are Recurrent Neural Networks?

- Consider now expressing this same relationship as

$$h_t = f\left(h_{t-1}, x_t; \theta\right)$$

  a process occasionally referred to as unfolding.
- You can think of h_{t-1} as an incomplete summary of the relevant information in the history of $x_{t-1}, \ldots, x_1$
- This formulation has tow important advantages: first, the model always has the same input size and second, the transition function f is the same at every time period which allows effective learning. (This is an example of parameter sharing between all the g_t functions).
- This relies on the assumption of stationarity: that the conditional distribution of the variables at time t given the variables at time $t-1$ does not change over time.

Simple Recurrent Neural Networks

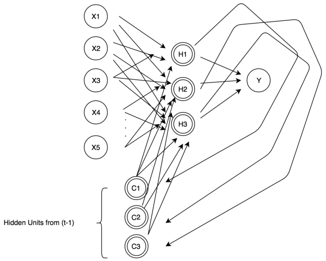
Elman and Jordan Networks

- Recurrent neural networks allow connections to go backwards: that is from neurons to other neurons or from the outputs back to hidden neurons.
- This creates a sense of memory as the network "remembers" previous outputs or states and allows such neural networks to work well with time series data.
- The two simplest recurrent networks are Elman and Jordan networks.
- An Elman network is simply an MLP with extra "context" layers; a context layer is simply the values of the same hidden layer carried over from the previous time period.
- A Jordan network is an MLP with an extra "context neuron" which is given by the network output at the previous time period.

Recurrent Networks

Elman and Jordan Networks

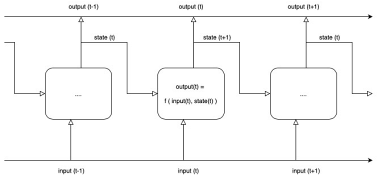
- An example of an Elman Neural Network



Recurrent Networks

A Simple RNN

- In the example below, the output of the RNN is a 2D tensor of shape (timesteps, output features). At each timestep t, the output of the RNN contains information about timesteps 1 to t. In many situations this is not necessary as you only need the last output at the end of the sequence



Deep Recurrent Neural Networks

Benefit and Challenges

- In the previous graph you can see that a RNN has three kinds of operations: (i) input to hidden unit, (ii) previous hidden unit to current hidden unit, and (iii) current hidden unit to the output.
- Would it be a good idea to add depth in any or all of these operations?
- The evidence suggests that the answer is yes. There are considerable challenges however.
- The main challenge is that using deep models to learn long term dependencies leads to the well known problem of vanishing or exploding gradient.
- Some early attempts to deal with this include echo-state networks and designing models that operate at multiple time scales.
- Multiple time scale models work in a number of ways: (i) adding direct connections from distant variables to present ones, (ii) having units with linear self connection and a weight near one on such connections ("leaky units"), and (iii) removing length-one connections and replacing them with longer ones.

Long Short Term Memory Models

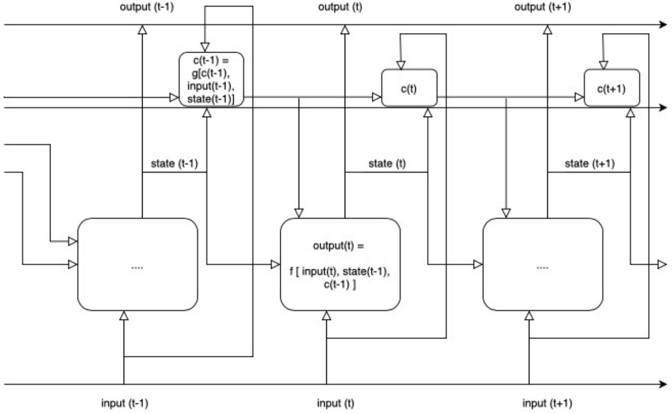
Introduction

- The most effective models capable of dealing with long term dependencies are Long Short Term Memory Models (LSTMs).
- These generalize the idea of the leaky unit to allow for weights that change over time.
- The LSTM setup allows the network to accumulate information over a long period of time but also to forget the old information once no longer relevant. Importantly, the model learns when to do this.

Recurrent Networks

An RNN with Memory

- Here is an example of an RNN with a memory cell, denoted by $c(t)$.



Long Short Term Memory Models

Long and Short Term Hidden Units

- The LSTM is based on the idea of creating a second kind of hidden unit, occasionally called the "cell state" or "memory state" and denoted here by s_i^t.
- This new unit captures the long term memory of the model and reflects the "L" in LSTM. This is complementary to the standard hidden unit h_i^t which we expect to see in a normal NN and which can be interpreted to capture the short term memory, reflecting the "S" in LSTM.
- As the model evolves over time, two kinds of hidden units are being estimated at each iteration:

1. The long term memory unit s_i^t (**L**STM)
2. The short term memory unit h_i^t (L**S**TM)

Long Short Term Memory Models

The Gates

- The LSTM is also based on the idea of "gating". A "gate" is fancy term to denote a weight that can take values from 0 to 1.
- Crucially in an LSTM these weights (the gates) have similar functional forms that the LSTM learns using the data. The sigmoid function is used to ensure the output is within the required 0 to 1 range.
- The role of the gates is to weigh information when we update (i) the long term memory unit by keeping old information, (iI) the long term memory unit by adding new information, and (iii) the short term memory unit.

Long Short Term Memory Models

The Gates

- Accordingly, there are three kinds of gates in an LSTM:

1. The **Forget Gate**:

the range of these functions is [0,1]
(sigmoid functions)

$$f_i^t = \sigma\big(b_i^f + \sum_j U_{i,j}^f x_j^t + \sum_j W_{i,j}^f h_j^{t-1}\big)$$

2. The **Input Gate**:

$$g_i^t = \sigma\big(b_i^g + \sum_j U_{i,j}^g x_j^t + \sum_j W_{i,j}^g h_j^{t-1}\big)$$

3. The **Output Gate**:

$$q_i^t = \sigma\big(b_i^q + \sum_j U_{i,j}^q x_j^t + \sum_j W_{i,j}^q h_j^{t-1}\big)$$

bias input weight hidden unit weight

Long Short Term Memory Models

The Gates

- There is a lot if information here so lets try to unpack these equations.
- First keep in mind that these are just weight functions; given the sigmoid transformation they are bound to produce an output in the $(0, 1)$ range.
- Each equation is indexed by time t and by hidden unit i. This means you will have one of these equations per hidden unit and each of them will be calculated at every time step.
- In the usual neural network fashion we have linear functions inside the sigmoid: each gate has its own bias, input weight and hidden unit weight. There are as many of these per gate as are hidden units (they are indexed by i) but they do not change over time (not indexed by t).

Long Short Term Memory Models

The Long Memory Hidden Unit

- The long term memory unit is updated via

$$s_i^t = f_i^t \times s_i^{t-1} + g_i^t \times \bar{s}_i^t$$

  where

$$\bar{s}_i^t = \sigma\big(b_i + \sum_j U_{i,j} x_j^t + \sum_j W_{i,j} h_j^{t-1}\big)$$

- The idea is that the forget gate allows the LSTM to keep some information from the previous state (recall a gate is a weight).
- Similarly, the input gate allows the LSTM to add new information, again weighted by its importance.
- The term $\bar{s}_i^t$ can be interpreted as the potential new information to be added to the long term memory.

Long Short Term Memory Models

The Short Memory Hidden Unit

- The short term memory unit is calculated via

$$h_i^t = q_i^t \times \tanh\left(s_i^t\right)$$

- The `tanh` function is a non-linear squashing function summarizing the long term memory unit and the output gate assigns the relevant importance by weighing the observation.

Long Short Term Memory Models

The Complete Model

- All the equations together for convenience:

$$f_i^t = \sigma\big(b_i^f + \sum_j U_{i,j}^f x_j^t + \sum_j W_{i,j}^f h_j^{t-1}\big)$$

$$g_i^t = \sigma\big(b_i^g + \sum_j U_{i,j}^g x_j^t + \sum_j W_{i,j}^g h_j^{t-1}\big)$$

$$q_i^t = \sigma\big(b_i^q + \sum_j U_{i,j}^q x_j^t + \sum_j W_{i,j}^q h_j^{t-1}\big)$$

$$s_i^t = f_i^t \times s_i^{t-1} + g_i^t \times \bar{s}_i^t$$

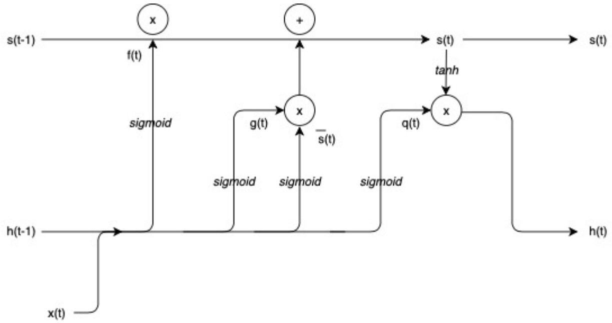
$$\bar{s}_i^t = \sigma\big(b_i + \sum_j U_{i,j} x_j^t + \sum_j W_{i,j} h_j^{t-1}\big)$$

$$h_i^t = q_i^t \times \tanh\left(s_i^t\right)$$

Recurrent Networks

An RNN with Memory

- Here is a graph of a simple LSTM.



Long Short Term Memory Models

- LSTMs have been shown to be more effective in learning long term dependencies than standard RNNs.
- Optimization for RNNs is usually done by some version of stochastic gradient descent.
- The input units $\bar{s}_i^t$ can have a non-sigmoid squashing function (e.g. tanh).
- However, all gates must use sigmoids to ensure that the estimated gate is in the interval of $(0, 1)$

Gated Recurrent Unit Models

- Gated Recurrent Unit (GRU) models are recurrent neural networks that are similar to LSTMs, though a little simpler and computationally lighter.
- The model equations are given by:

$$f_i^t = \sigma\big(b_i^f + \sum_j U_{i,j}^f x_j^t + \sum_j W_{i,j}^f h_j^{t-1}\big)$$

$$r_i^t = \sigma\big(b_i^r + \sum_j U_{i,j}^r x_j^t + \sum_j W_{i,j}^r h_j^{t-1}\big)$$

$$\bar{h}_i^t = \tanh\big(b_i + \sum_j U_{i,j} x_j^t + \sum_j W_{i,j} r_i^t h_j^{t-1}\big)$$

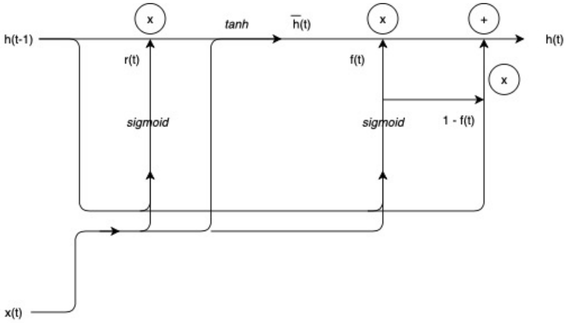
$$h_i^t = (1 - f_i^t) \times h_i^{t-1} + f_i^t \times \bar{h}_i^t$$

- The main difference between an LSTM and a GRU is that a single gate (the f_i^t above) controls both the forgetting and the updating.

Gated Recurrent Unit Models

A GRU

- Here is a graph of a simple GRU.



Bi-directional RNNs

- RNNs are order dependent; they process the timesteps of the input in order and shuffling that order will change the results.
- A bi-directional RNN (BRNN) consists of two RNNs that process the data in order and in reverse order respectively and then merger the representations.
- This works well for text problems because the importance of a word in understanding the sentence is not dependent on the position of that word in the sentence.
- This is a general idea in machine learning; representations that are different yet useful are worth exploring, and the more they differ the better.
- BRNNS are not expected to do well in a time series forecasting problem, where exact chronological order matters.

Simple RNNs

- In Keras this is implemented as follows:

```
# Pseudo code
model<- keras_model_sequential() %>%
layer_simple_rnn(units = 32, return_sequences = TRUE, input_shape = ...)
```

- This processes batches of sequences like other Keras layers which means that it takes inputs of shape (batch size, timesteps, input features).
- Also like all recurrent layers in Keras it can return either (i) the full suequence of outputs for each timestep; a 3D tensor of shape (batch size, timesteps, output features) or (ii) only the last output for each input sequence; a 2D tensor of shape (batch size, output features)).
- These two modes are controlled by the `return_sequences` argument.

Stacked Simple RNNs

- It is sometimes useful to stack several recurrent layers to increase the representational power of the model.
- To do this we need to have all the intermediate layers to return full sequences.

```r
# Pseudo code
model<- keras_model_sequential() %>%
  layer_simple_rnn(units = 32, return_sequences = TRUE, input_shape = ...) %>%
  layer_simple_rnn(units = 32, return_sequences = TRUE) %>%
  layer_simple_rnn(units = 32, return_sequences = TRUE) %>%
  layer_simple_rnn(units = 32, return_sequences = FALSE)
```

LSTMs in Practice

- Now lets use an LSTM to forecast temperature readings (following the example in the book "Deep Learning with R" by Franois Chollet with J.J. Allaire).
- In the dataset below 14 features (e.g. air temperature, humidity, etc.) were recorded every 10 minutes over several years (2003-2016) and will be used to predict air temperature 24 hours in the future.

```r
# First lets download and uncompress the data.
dir.create("~/Downloads/jena_climate", recursive = TRUE)
download.file(
  "https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip",
  "~/Downloads/jena_climate/jena_climate_2009_2016.csv.zip"
)
unzip(
  "~/Downloads/jena_climate/jena_climate_2009_2016.csv.zip",
  exdir = "~/Downloads/jena_climate"
)
```

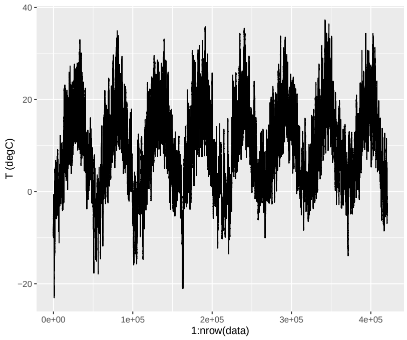
LSTMs in Practice

- You can use the following code to take a closer look at the data:

```
library(keras); library(readr); library(ggplot2); library(magrittr)
data_dir <- "~/Downloads/jena_climate"
fname <- file.path(data_dir, "jena_climate_2009_2016.csv")
data <- read_csv(fname)

## Parsed with column specification:
## cols(
##  `Date Time` = col_character(),
##  `p (mbar)` = col_double(),
##  `T (degC)` = col_double(),
##  `Tpot (K)` = col_double(),
##  `Tdew (degC)` = col_double(),
##  `rh (%)` = col_double(),
##  `VPmax (mbar)` = col_double(),
##  `VPact (mbar)` = col_double(),
##  `VPdef (mbar)` = col_double(),
##  `sh (g/kg)` = col_double(),
##  `H2OC (mmol/mol)` = col_double(),
##  `rho (g/m**3)` = col_double(),
##  `wv (m/s)` = col_double(),
##  `max. wv (m/s)` = col_double(),
##  `wd (deg)` = col_double()
## )

tibble::glimpse(data)

## Observations: 420,551
## Variables: 15
## $ `Date Time`       <chr> "01.01.2009 00:10:00", "01.01.2009 00:20:00", "01...
```

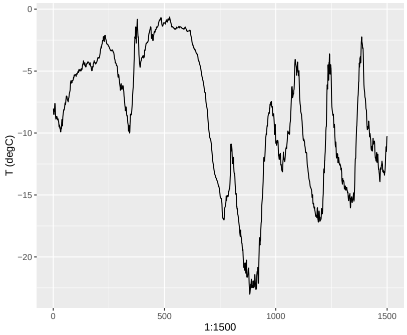
LSTMs in Practice

```r
#plot the data
ggplot(data, aes(x = 1:nrow(data), y = `T (degC)`)) + geom_line()
```



LSTMs in Practice

```
#plot the data
ggplot(data[1:1500, ], aes(x = 1:1500, y = `T (degC)`)) + geom_line()
```



LSTMs in Practice

- We can formulate the problem as follows: given data going as far back as `lookback` timesteps (e.g. 10 minutes above) and sampled every `steps` timesteps, can we make a prediction `delay` timesteps ahead? In this example we will use:
- `lookback` = 1440 (observations will go back 10 days)
- `steps` = 6 (observations sampled at one data point per hour)
- `delay` = 144 (target will be 24 hours in the future)

LSTMs in Practice

- To get started we will do two things: (i) we will normalize each timeseries independently so that they are all on a similar scale, and (ii) create a *generator* function that takes the current array of data and yield batches of data (features and targets).
- To normalize the data we will subtract the mean of each timeseries and divide by its standard deviation.

```r
# Convert a Data Frame to a Numeric Matrix
data <- data.matrix(data[, -1])
train_data <- data[1:200000, ]
mean <- apply(train_data, 2, mean)
std  <- apply(train_data, 2, sd)
data <- scale(data ,center = mean, scale = std)
```

LSTMs in Practice

- A generator function is a special type of function that you call repeatedly to obtain a sequence of values. In essence generators are functions that work as *iterators*. The following gives an example:

```
simple_generator <- function(start) {
  value <- start - 1
  function () {
    value <<- value + 1 # note the global assignment here
    value
  }
}
gen <- simple_generator(10) # assigns the function to gen
gen()

## [1] 10

gen()

## [1] 11
```

LSTMs in Practice

- The current state of the generator is the `value` variable, defined outside the function. Note that genertator functions passed to Keras training methods (such as `fit_generator()`) should always return values infinitely. The number of calls to the generator function is controlled by the `epochs` and `steps_per_epoch` arguments.

LSTMs in Practice

```
model <- keras_model_sequential() %>%
  layer_gru(units = 32,
            dropout = 0.2,
            recurrent_dropout = 0.2,
            input_shape = list(NULL, dim(data)[[-1]])) %>%
  layer_dense(units = 1)
model %>% compile(
  optimizer = optimizer_rmsprop(),
  loss = "mae"
)

history <- model %>% fit_generator(
  train_gen,
  steps_per_epoch = 100,
  epochs = 5,
  validation_data = val_gen,
  validation_steps = val_steps
)
```

Acknowledgements

These notes are based on and follow closely the excellent book "Deep Learning with R" by Franois Chollet (the creator of Keras) with J.J. Allaire.

These notes are based on also follow closely Deep Learning by I. Goodfellow, Y. Bengio and A. Courville.

Additionally, you can refer to the excellent books for more examples and details: "The Deep Learning with R" and "Deep Learning with Python", the latter also by Franois Chollet.

A Deep Learning with Keras cheat sheet is available here `https://github.com/rstudio/cheatsheets/raw/master/keras.pdf`. A great blog about LSTMs can be found here: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`