RNNs for Timeseries Analysis

Bruno Gonçalves

www.bgoncalves.com
github.com/bmtgoncalves/RNN
The views and opinions expressed in this article are those of the authors and do not necessarily reflect the official policy or position of my employer. The examples provided with this tutorial were chosen for their didactic value and are not mean to be representative of my day to day work.
References

- *Pattern Recognition and Machine Learning* by Christopher M. Bishop
- *Machine Learning: A Probabilistic Perspective* by Kevin P. Murphy
- *Deep Learning with Python* by François Chollet
- *Deep Learning with Keras* by Richard S. S. Thompson
- *Mastering TensorFlow 1.x* by Pablo Castellanos

@bgoncalves  
www.bgoncalves.com
How the Brain “Works” (Cartoon version)
How the Brain “Works” (Cartoon version)

- Each neuron receives input from other neurons
- $10^{11}$ neurons, each with $10^4$ weights
- Weights can be positive or negative
- Weights adapt during the learning process
- “neurons that fire together wire together” (Hebb)
- Different areas perform different functions using same structure (Modularity)
How the Brain “Works” (Cartoon version)
Optimization Problem

• (Machine) Learning can be thought of as an optimization problem.

• Optimization Problems have 3 distinct pieces:
  
  • The **constraints**
  
  • The **function** to optimize
  
  • The **optimization algorithm**

  Neural Network
  
  Prediction Error
  
  Gradient Descent
Artificial Neuron

\[ w^T x \]

\[ \phi(z) \]

Inputs  Weights  Activation function  Output

@bgoncalves  www.bgoncalves.com
Activation Function - Sigmoid

- Non-Linear function
- Differentiable
- non-decreasing
- Compute new sets of features
- Each layer builds up a more abstract representation of the data
- Perhaps the most common

\[ \phi(z) = \frac{1}{1 + e^{-z}} \]
Activation Function - tanh

- Non-Linear function
- Differentiable
- non-decreasing
- Compute new sets of features
- Each layer builds up a more abstract representation of the data

\[ \phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \]
Forward Propagation

• The output of a perceptron is determined by a sequence of steps:
  • obtain the inputs
  • multiply the inputs by the respective weights
  • calculate output using the activation function

• To create a multi-layer perceptron, you can simply use the output of one layer as the input to the next one.

• But how can we propagate back the errors and update the weights?
Backward Propagation of Errors (BackProp)

• BackProp operates in two phases:
  
  • Forward propagate the inputs and calculate the deltas
  
  • Update the weights

• The error at the output is a weighted average difference between predicted output and the observed one.

• For inner layers there is no “real output”!
Loss Functions

- For learning to occur, we must quantify how far off we are from the desired output. There are two common ways of doing this:

  - Quadratic error function:
    \[ E = \frac{1}{N} \sum_n (y_n - a_n)^2 \]

  - Cross Entropy
    \[ J = -\frac{1}{N} \sum_n \left[ y_n^T \log a_n + (1 - y_n)^T \log (1 - a_n) \right] \]

  The Cross Entropy is complementary to sigmoid activation in the output layer and improves its stability.
Gradient Descent

- Find the gradient for each training batch

- Take a step downhill along the direction of the gradient

\[ \theta_{mn} \leftarrow \theta_{mn} - \alpha \frac{\partial H}{\partial \theta_{mn}} \]

- where \( \alpha \) is the step size.

- Repeat until "convergence".
**INPUT TERMS**
- Features
- Predictions
- Attributes
- Predictable Variables

**MACHINE**
- Algorithms
- Techniques
- Models

**OUTPUT TERMS**
- Classes
- Responses
- Targets
- Dependant Variables
Feed Forward Networks

\[ h_t = f(x_t) \]
Feed Forward Networks

\[ h_t = f(x_t) \]
Feed Forward Networks

\[ h_t = f(x_t) \]
Recurrent Neural Network (RNN)

\[ h_t = f(x_t, h_{t-1}) \]
Recurrent Neural Network (RNN)
Recurrent Neural Network (RNN)

- Each output depends (implicitly) on all previous outputs.

- Input sequences generate output sequences (seq2seq)
Recurrent Neural Network (RNN)

\[ h_t = \tanh \left( W h_{t-1} + U x_t \right) \]
Recurrent Neural Network (RNN)

\[ h_t = \tanh \left( W h_{t-1} + U x_t \right) \]

Concatenate both inputs.
Timeseries

• Temporal sequence of data points

• Consecutive points are strongly correlated

• Common in statistics, signal processing, econometrics, mathematical finance, earthquake prediction, etc

• Numeric (real or discrete) or symbolic data
Long-Short Term Memory (LSTM)

- What if we want to keep explicit information about previous states (memory)?

- How much information is kept, can be controlled through gates.

- LSTMs were first introduced in 1997 by Hochreiter and Schmidhuber.
Long-Short Term Memory (LSTM)

\[ f = \sigma \left( W_f h_{t-1} + U_f x_t \right) \]
\[ g = \tanh \left( W_g h_{t-1} + U_g x_t \right) \]
\[ i = \sigma \left( W_i h_{t-1} + U_i x_t \right) \]
\[ c_t = (c_{t-1} \otimes f) + (g \otimes i) \]
\[ o = \sigma \left( W_o h_{t-1} + U_o x_t \right) \]
\[ h_t = \tanh \left( c_t \otimes o \right) \]
Long-Short Term Memory (LSTM)

\[
\begin{align*}
    f &= \sigma \left( W_f h_{t-1} + U_f x_t \right) \\
    i &= \sigma \left( W_i h_{t-1} + U_i x_t \right) \\
    o &= \sigma \left( W_o h_{t-1} + U_o x_t \right) \\
    g &= \tanh \left( W_g h_{t-1} + U_g x_t \right) \\
    c_t &= (c_{t-1} \otimes f) + (g \otimes i) \\
    h_t &= \tanh (c_t) \otimes o
\end{align*}
\]

Forget gate: How much of the previous state should be kept?
Long-Short Term Memory (LSTM)

Input gate: How much of the previous output should be remembered?

\[ f = \sigma \left( W_f h_{t-1} + U_f x_t \right) \]
\[ i = \sigma \left( W_i h_{t-1} + U_i x_t \right) \]
\[ o = \sigma \left( W_o h_{t-1} + U_o x_t \right) \]
\[ g = \tanh \left( W_g h_{t-1} + U_g x_t \right) \]
\[ c_t = \left( c_{t-1} \otimes f \right) + \left( g \otimes i \right) \]
\[ h_t = \tanh \left( c_t \right) \otimes o \]
Long-Short Term Memory (LSTM)

Output gate: How much of the previous output should contribute?

\[
\begin{align*}
    f &= \sigma(W_f h_{t-1} + U_f x_t) \\
    i &= \sigma(W_i h_{t-1} + U_i x_t) \\
    o &= \sigma(W_o h_{t-1} + U_o x_t) \\
    c_t &= (c_{t-1} \otimes f) + (g \otimes i) \\
    g &= \tanh(W_g h_{t-1} + U_g x_t) \\
    h_t &= \tanh(c_t) \otimes o
\end{align*}
\]

All gates use the same inputs and activation functions, but different weights.

Element wise addition
Element wise multiplication
1 minus the input
Long-Short Term Memory (LSTM)

Output gate: How much of the previous output should contribute?

\[ f = \sigma \left( W_f h_{t-1} + U_f x_t \right) \]
\[ i = \sigma \left( W_i h_{t-1} + U_i x_t \right) \]
\[ o = \sigma \left( W_o h_{t-1} + U_o x_t \right) \]
\[ c_t = (c_{t-1} \otimes f) + (g \otimes i) \]
\[ g = \tanh \left( W_g h_{t-1} + U_g x_t \right) \]
\[ h_t = \tanh \left( c_t \right) \otimes o \]
Long-Short Term Memory (LSTM)

$$f = \sigma \left( W_f h_{t-1} + U_f x_t \right) \quad g = \tanh \left( W_g h_{t-1} + U_g x_t \right)$$

$$i = \sigma \left( W_i h_{t-1} + U_i x_t \right) \quad c_t = \left( c_{t-1} \otimes f \right) + \left( g \otimes i \right)$$

$$o = \sigma \left( W_o h_{t-1} + U_o x_t \right) \quad h_t = \tanh \left( c_t \right) \otimes o$$

Element wise addition
Element wise multiplication
1 minus the input
Long-Short Term Memory (LSTM)

\[
\begin{align*}
    f &= \sigma \left( W_f h_{t-1} + U_f x_t \right) \\
    i &= \sigma \left( W_i h_{t-1} + U_i x_t \right) \\
    o &= \sigma \left( W_o h_{t-1} + U_o x_t \right) \\
    c_t &= \left( c_{t-1} \otimes f \right) + \left( g \otimes i \right) \\
    g &= \tanh \left( W_g h_{t-1} + U_g x_t \right) \\
    h_t &= \tanh \left( c_t \right) \otimes o
\end{align*}
\]

Output:
Combine all available information.
Using LSTMs

Sequence Length

#features

W

LSTM

LSTM

inputs

inputs

inputs

inputs

inputs

#LSTM cells

σ
Applications

• Language Modeling and Prediction

• Speech Recognition

• Machine Translation

• Part-of-Speech Tagging

• Sentiment Analysis

• Summarization

• Time series forecasting
Gated Recurrent Unit (GRU)

• Introduced in 2014 by Cho

• Meant to solve the Vanishing Gradient Problem

• Can be considered as a simplification of LSTMs

• Similar performance to LSTM in some applications, better performance for smaller datasets.
Gated Recurrent Unit (GRU)

\[
\begin{align*}
z &= \sigma \left( W_z h_{t-1} + U_z x_t \right) \\
r &= \sigma \left( W_r h_{t-1} + U_r x_t \right) \\
\sigma &= \text{Element wise addition} \\
\times &= \text{Element wise multiplication} \\
1-1 &= 1 \text{ minus the input} \\
c &= \tanh \left( W_c \left( h_{t-1} \otimes r \right) + U_c x_t \right) \\
h_t &= (z \otimes c) + \left( (1 - z) \otimes h_{t-1} \right)
\end{align*}
\]
Gated Recurrent Unit (GRU)

Update gate: How much of the previous state should be kept?

\[
\begin{align*}
    z &= \sigma \left( W_z h_{t-1} + U_z x_t \right) \\
    r &= \sigma \left( W_r h_{t-1} + U_r x_t \right) \\
    c &= \tanh \left( W_c (h_{t-1} \otimes r) + U_c x_t \right) \\
    h_t &= (z \otimes c) + \left( (1 - z) \otimes h_{t-1} \right)
\end{align*}
\]
Gated Recurrent Unit (GRU)

Reset gate: How much of the previous output should be removed?

\[
\begin{align*}
z &= \sigma \left( W_z h_{t-1} + U_z x_t \right) \\
r &= \sigma \left( W_r h_{t-1} + U_r x_t \right) \\
c &= \tanh \left( W_c \left( h_{t-1} \otimes r \right) + U_c x_t \right) \\
h_t &= (z \otimes c) + \left( (1 - z) \otimes h_{t-1} \right)
\end{align*}
\]
Gated Recurrent Unit (GRU)

\[ h_t = (z \otimes c) + ((1 - z) \otimes h_{t-1}) \]

Current memory: What information do we remember right now?

\[ z = \sigma \left( W_z h_{t-1} + U_z x_t \right) \]
\[ r = \sigma \left( W_r h_{t-1} + U_r x_t \right) \]
\[ c = \tanh \left( W_c \left( h_{t-1} \otimes r \right) + U_c x_t \right) \]
Gated Recurrent Unit (GRU)

Output: Combine all available information.

\[ z = \sigma \left( W_z h_{t-1} + U_z x_t \right) \]
\[ r = \sigma \left( W_r h_{t-1} + U_r x_t \right) \]
\[ c = \tanh \left( W_c \left( h_{t-1} \otimes r \right) + U_c x_t \right) \]
\[ h_t = (z \otimes c) + ((1 - z) \otimes h_{t-1}) \]
Neural Networks?

- $c_{t-1}$
- $h_{t-1}$
- $x_t$
- $c_t$
- $h_t$
Or legos?

https://keras.io

@bgoncalves

www.bgoncalves.com
Keras

- **Open Source** neural network library written in **Python**

- **TensorFlow**, Microsoft Cognitive Toolkit or Theano backends

- Enables **fast experimentation**

- Created and maintained by **François Chollet**, a Google engineer.

- Implements **Layers**, **Objective/Loss functions**, **Activation functions**, **Optimizers**, etc…
Keras

- **keras.models.Sequential(layers=None, name=None)** - is the workhorse. You use it to build a model layer by layer. Returns the object that we will use to build the model.

- **keras.layers**
  - **Dense(units, activation=None, use_bias=True)** - None means linear activation. Other options are, ’tanh’, ’sigmoid’, ’softmax’, ’relu’, etc.
  - **Dropout(rate, seed=None)**
  - **Activation(activation)** - Same as the activation option to Dense, can also be used to pass TensorFlow or Theano operations directly.
  - **SimpleRNN(units, input_shape, activation='tanh', use_bias=True, dropout=0.0, return_sequences=False)**
  - **GRU(units, input_shape, activation='tanh', use_bias=True, dropout=0.0, return_sequences=False)**
  - **LSTM(units, input_shape, activation='tanh', use_bias=True, dropout=0.0, return_sequences=False)**
Keras

- `model = Sequential()`

- `model.add(layer)` - Add a layer to the top of the model

- `model.compile(optimizer, loss)` - We have to compile the model before we can use it
  - `optimizer` - ‘adam’, ‘sgd’, ‘rmsprop’, etc...
  - `loss` - ‘mean_squared_error’, ‘categorical_crossentropy’, ‘kullback_leibler_divergence’, etc...

- `model.fit(x=None, y=None, batch_size=None, epochs=1, verbose=1, validation_split=0.0, validation_data=None, shuffle=True)`

- `model.predict(x, batch_size=32, verbose=0)` - fit/predict interface similar to sklearn.

- `model.summary()` - Output a textual representation of the model

https://keras.io
github.com/bmtgoncalves/RNN