Time Series Analysis

7. Deep Learning for Time Series

Andrew Lesniewski

Baruch College
New York

Fall 2023

Outline

1. Feedforward Neural Networks

2. Recurrent Neural Networks

3. LSTM

Artificial neural networks

- *Artificial neural networks* (ANN) are approximation methodologies for computing highly complex functions, whose designs are inspired by biological neural networks.
- These approximations are built by aggregating a number of basic *activation functions* $\varphi(x)$, which are mathematical models of the rate of action potential firing (output) by the (biological) neuron.
- An activation function is usually assumed to be continuous, its range may or may not be bounded.
- The choice of activation functions depends on the output variables of the problem.
- Each ANN comes with its *architecture*, i.e. design choices of the activation functions and how they interact with each other.
- The process of estimating the parameters of an ANN is referred to as *learning* or *training* the network.

Artificial neural networks

- The simplest activation function is the (discontinuous) *binary* activation

$$\varphi(x) = \begin{cases} 0, & \text{if } x < 0, \\ 1, & \text{if } x \geq 0. \end{cases} \tag{1}$$

- A standard example of a smooth activation function is the *logistic* (a.k.a. *sigmoid*) activation function given by

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \,. \tag{2}$$

- Another important example of a smooth activation function is the `tanh` activation:

$$\varphi(x) = \tanh(x). \tag{3}$$

- A frequently used continuous (but not smooth) activation function is the *rectified linear unit* (ReLU) activation:

$$\varphi(x) = \begin{cases} 0, & \text{if } x < 0, \\ x, & \text{if } x \geq 0. \end{cases} \tag{4}$$

Single-layer perceptron

- The simplest ANN is a *single-layer perceptron*, which consists of n neurons.
- The term perceptron often refers to networks consisting of just one neuron.
- The *inputs* $x_1, \ldots, x_m$ to the k-th neuron are combined through a series of *weights* $w_{k1}, \ldots, w_{km}$ and a *bias* b_k, and are fed directly to the *output* y_k through the formula:
  $$y_k = \varphi(a_k), \tag{5}$$
  where the activation a is defined by
  $$a_k = \sum_{i=1}^{m} w_{ki} x_i + b_k. \tag{6}$$

Single-layer perceptron

- The set of inputs to a perceptron forms the *input layer*, while the outputs $y_1, \ldots, y_n$ form the (single) *output layer*.
- A single-layer perceptron with logistic activation is identical to the logistic regression model.
- The derivative of $\sigma(x)$ satisfies the differential equation

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)), \tag{7}$$

which makes training of the network easy.

Single-layer perceptron

- Graphically, this is represented as follows:

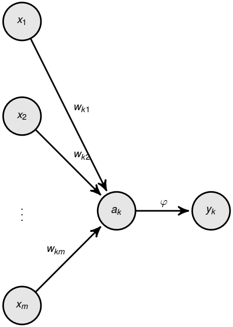


Figure: Structure of an RNN with a single hidden layer

Multi-layer perceptron

- Single-layer perceptrons are only capable of learning a limited class of patterns (namely, linearly separable patterns).
- It is, for example, impossible for a single-layer perceptron network to learn an XOR (exclusive or) function.
- Composing activation functions multiple number of times, and thus creating intermediate or *hidden* layers, leads to the concept of a *multi-layer perceptron* (MLP).
- An MLP consists of three or more layers of nonlinearly activating nodes: an input layer, an output layer, and one or more hidden layers.
- MLPs are fully connected: each node in one layer connects with a certain weight w_{ki} to every node in the following layer.

Multi-layer perceptron

- Below is the graphical representation of an example of an MLP:

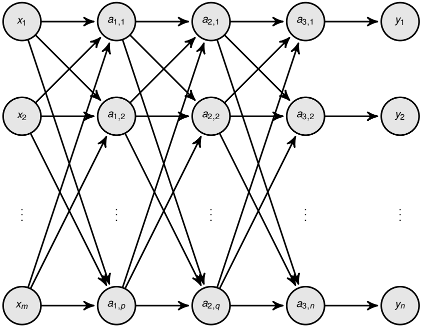


Figure: Structure of an RNN with a single hidden layer

Multi-layer perceptron

- In this network, the activations and outputs are calculated as follows:

$$\begin{aligned} a_{1,i} &= \varphi_1((W_1 x)_i + b_{1,i}), \\ a_{2,i} &= \varphi_2((W_2 a_1)_i + b_{2,i}), \\ a_{3,i} &= \varphi_3((W_3 a_2)_i + b_{3,i}), \\ y_i &= a_{3,i}, \end{aligned} \tag{8}$$

where W_k are the matrices of weights for layer k, and b_k are the vectors of biases for layer k, respectively.

Multi-layer perceptron

- An MLP is an example of a *feedforward network*.
- A feedforward neural network is the simplest type of an ANN.
- In a feedforward ANN, the information moves only in the forward direction, namely from the input layer through the hidden layers (if any) to the output layer. There are no cycles in the network.
- A two-layer neural network is capable of calculating XOR.
- In fact, it can be shown that MLPs are capable of learning any boolean function.
- The theoretical underpinning of ANNs is the fundamental *universal approximation theorem*.
- It states that any continuous function mapping a bounded interval in $\mathbb{R}^n$ into a bounded interval can be approximated arbitrarily closely by an MLP with just one hidden layer.

Universal approximation theorem

- Early versions of this theorem, with the sigmoid activation function, were formulated in the 1980s and 1990s by George Cybenko and Kurt Hornik. These results were later shown to hold for a wide range range of activation functions.
- Allan Pinkus showed in 1999 [2] that the universal approximation property is equivalent to working with an activation function which is *not* a polynomial.
- The following statement is the most useful version of the universal approximation property.
- *Let φ be a continuous (activation) function. For $x \in \mathbb{R}^n$ we let $\varphi(x)$ denote a vector in $\mathbb{R}^n$ with components $\varphi(x)_i = \varphi(x_i)$ (i.e. φ is applied to each component of x). Let $f : R^n \to \mathbb{R}^m$ be a continuous function. Then φ is not polynomial, if and only if for every such function f, any compact a subset $K \subseteq \mathbb{R}^n$, and any $\varepsilon > 0$, there exist: (i) an integer $k \in \mathbb{N}$, (ii) a matrix of weights $W \in \mathrm{Mat}_{k,n}(\mathbb{R})$, (iii) a vector of biases $b \in \mathbb{R}^k$, and (iv) a matrix $C \in \mathrm{Mat}_{m,k}(\mathbb{R})$ such that*

$$\max_{x \in K} \|f(x) - C\varphi(Wx + b)\| < \varepsilon.$$

Training a multi-layer perceptron

- Neural networks are trained by presenting them with data $d_1, \ldots, d_N$ and selecting the weights and biases, collectively denoted by θ, in such a way that a measure of distance between $d_1, \ldots, d_N$, the *loss function*, and the outputs of the network are minimized.
- The choice of the loss function is motivated by the problem at hand.
- For example, it can be the sum of squared errors,

$$L(\theta) = \frac{1}{2} \sum_{i=1}^{N} \left(y_i(\theta) - d_i\right)^2, \tag{9}$$

  leading to a nonlinear least square problem.
- Optimization techniques used to minimize this loss function usually rely on the *gradient descent method*, which requires evaluating first derivatives of the loss function.
- These derivatives are calculated recursively using the chain rule of multivariate calculus.

Training a multi-layer perceptron

- *Backpropagation* is the algorithm that organizes this calculation in an optimal way.
- Backpropagation can only be applied on networks with differentiable activation functions, such as the sigmoid or tanh.
- For efficiency, it requires the knowledge of the derivatives of activation functions. *Automatic differentiation* is a technique that provides the derivatives to the training algorithm.
- The gradient descent method is an iterative algorithm and it continues until a termination criterion is met. At each iteration the algorithm adjusts the weights and biases at each node.
- If, after iterating through this process, the network converges to a state where the loss function is small, we say that the network has learned the target function.

RNNs

- *Recurrent neural networks* (RNN), are a class of neural networks that allow outputs from previous layers to be used as inputs.
- RNNs are suitable for processing sequential data, such as speech recognition, word processing, time series analysis, etc.
- RNNs are frequently written in terms of a state space equation:

$$h^{(t)} = F(h^{(t-1)}, x^{(t)}|\theta), \tag{10}$$

where $h^{(t)}$ is the state variable, and $x^{(t)}$ is the input series.

RNNs

- The graph below shows an example of an RNN with a single hidden layer:

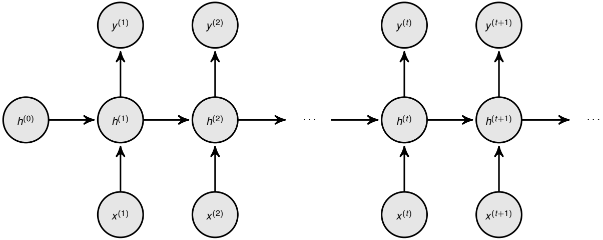


Figure: Elman network

RNNs

- The vertical directions in this graph are feedforward networks with one hidden layer.
- One can think of this graph as the result of “unfolding” the recursion (10).
- At each time step t, the activation $a^{(t)}$ and output $y^{(t)}$ are given by:

$$h^{(t)} = \varphi_1\big(W_{hh}h^{(t-1)} + W_{hx}x^{(t)} + b_h\big), \tag{11}$$

and

$$y^{(t)} = \varphi_2\big(W_{yh}h^{(t)} + b_y\big) \tag{12}$$

respectively, where $\varphi_1(x)$ and $\varphi_2(x)$ are (possibly different) activation functions.

- Note that the weight matrices W_{hh}, W_{hx}, W_{yh}, and bias vectors b_a, b_y are shared across different time steps.
- This RNN is known as an *Elman network*.

RNNs

- A *Jordan network* is defined by

$$\begin{aligned} h^{(t)} &= \sigma\big(W_{hy}y^{(t-1)} + W_{hx}x^{(t)} + b_h\big),\\ y^{(t)} &= \sigma\big(W_{yh}h^{(t)} + b_y\big). \end{aligned} \tag{13}$$

- It can garphically be unfolded as follows:

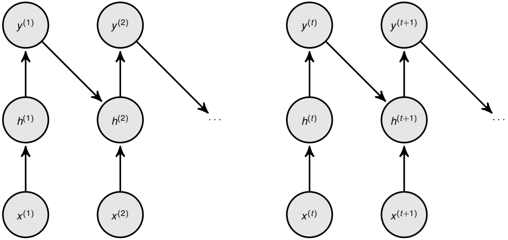


Figure: Jordan network

DRNN

- Many other RNN architectures have been used in various applications.
- *Deep recurrent neural networks* (DRNN) are architectures involving multiple hidden layers for each time step t.
- Below is the graphical representation of a DRNN.

DRNN

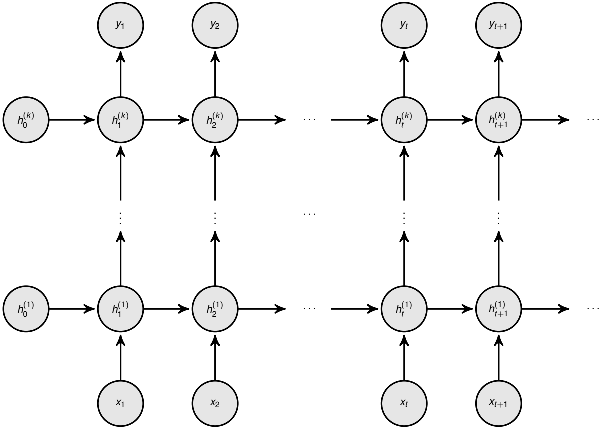


Figure: Structure of a deep RNN

Backpropagation through time

- The loss function of an RNN is a sum of loss functions corresponding to each time step t:
$$L = \sum_{t=1}^{T} L_t . \tag{14}$$
- For example, if we maximize the likelihood of the data set $\hat{y}_1, \ldots, \hat{y}_T$, then we could have
$$L_t = - \log \mathcal{L}(\theta \,|\, \hat{y}_1, \ldots, \hat{y}_t), \tag{15}$$
where $\mathcal{L}$ denotes the likelihood function.
- The backward propagation algorithm can be applied to (14):
$$\nabla_\theta L = \sum_{t=1}^{T} \nabla_\theta L_t . \tag{16}$$
- This approach leads to the *backpropagation through time* (BPTT) algorithm.

Backpropagation in time

- In principle, basic RNNs can keep track of arbitrary long-term time dependencies in the input series.
- In practice, however, one faces the following computational issue.
- When training an RNN over a long time period using BPTT, the gradients of the loss function can “vanish” (i.e. become very small) or “explode” (i.e. become very large).
- This is a result of the multiplicative nature of backpropagation: computational errors due to the finite precision arithmetic used by computers tend to accumulate multiplicatively.
- One can frequently solve the exploding gradients problem by *gradient clipping*, i.e. suitably capping the value of $\|\nabla L\|$, and thus preventing it from exploding.
- A number of approaches have been proposed to partially mitigate the vanishing gradient problem.

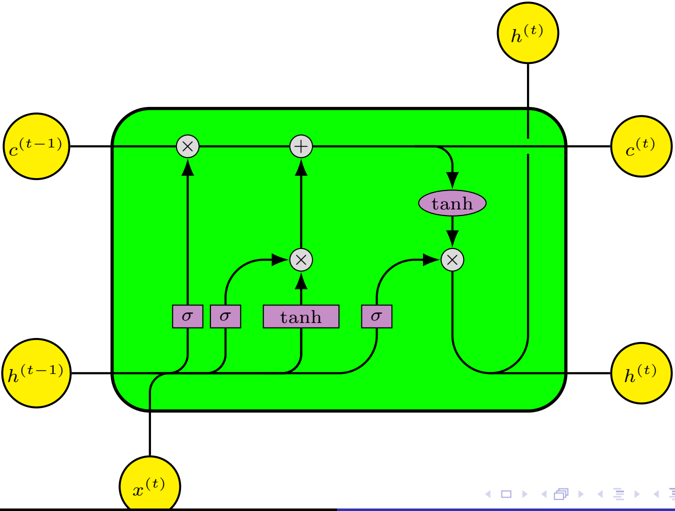
LSTM

- An improved version of RNNs, namely the *Long Short Term Memory* (LSTM) model, was proposed in [5].
- LSTM solves the problem of vanishing gradients by controlling the flow of information through a node of the network.
- This is achieved by replacing the hidden nodes of an RNN with carefully designed *LSTM units*.
- An LSTM unit protects its hidden activation, called the cell state, by using additional “gates”.
- LSTM networks may still exhibit exploding gradients.
- There exists a large number of versions of the LSTM. Another popular design, somewhat simpler than LSTM, is the *Gated Recurrent Unit* (GRU).

LSTM

- A common LSTM architecture is composed of:
  - (i) a *cell*, the memory part of the LSTM unit, responsible for keeping track of the dependencies between the elements in the input sequence,
  - (ii) an *input gate*, controls the extent to which a new value flows into the cell,
  - (iii) a *forget gate*, controls the extent to which a value remains in the cell,
  - (iv) an *output gate*, controls the extent to which the value in the cell is used to compute the output activation of the LSTM unit.
- The LSTM gates gates are connected among each other.
- The weights and biases of these connections are learned during training.
- LSTM are computationally significantly more costly than conventional RNNs.

LSTM


$h^{(t)}$
$c^{(t-1)}$
$\times$
$+$
$c^{(t)}$
tanh
$\times$
$\times$
σ
σ
tanh
σ
$h^{(t-1)}$
$h^{(t)}$
$x^{(t)}$

LSTM

- The protected cell activation at time step t is denoted by $c^{(t)}$, whereas $h^{(t)}$ is the (hidden) activation that will be passed on to other components of the model.
- The equations defining LSTM are as follows:

$$\begin{aligned}
f^{(t)} &= \sigma(W_{fh}h^{(t-1)} + W_{fx}x^{(t)} + b_f), \\
i^{(t)} &= \sigma(W_{ih}h^{(t-1)} + W_{ix}x^{(t)} + b_i), \\
c'^{(t)} &= \tanh(W_{ch}h^{(t-1)} + W_{cx}x^{(t)} + b_c), \\
c^{(t)} &= f^{(t)}c^{(t-1)} + i^{(t)}c'^{(t)} \\
o^{(t)} &= \sigma(W_{oh}h^{(t-1)} + W_{oc}c^{(t)} + b_o), \\
h^{(t)} &= o^{(t)}\sigma(c^{(t)}),
\end{aligned} \tag{17}$$

  where $f^{(t)}$, $i^{(t)}$, and $o^{(t)}$ are the activations of the input, forget, and output gates at t.
- These equations can be understood as follows.

LSTM

- The forget gate takes the information from the current input and previous hidden state, and decides what information should be discarded or retained.
- Since $\sigma(x)$ interpolates between 0 and 1, the values of $f^{(t)}$ range between these values.
- The closer $f^{(t)}$ to 0, the smaller is the contribution of $c^{(t-1)}$ to $c^{(t)}$. A value close to 1 means keeping $c^{(t-1)}$.
- The input gate updates the cell state it two steps:
  - (i) We pass the previous hidden state and current input into the sigmoid activation.
  - (ii) We also pass the previous hidden state and current input into the `tanh` activation. The result is multiplied by the output of the sigmoid. This determines the extent to which the output of the `tanh` function will impact the update of the cell state (depending on how close $i^{(t)}$ is to 0 or 1).

LSTM

- We now use these values to update the cell state. The cell state is multiplied (pointwise) by the forget vector.
- The updated cell state is the sum of this vector and the vector calculated in Step (ii) above.
- The output gate controls what the next hidden state should be:
  - (i) We pass the previous hidden state and the current input to the sigmoid activation.
  - (ii) Then we pass the updated cell state and the previous hidden state to the tanh activation.
- We multiply the outputs of the two activation functions to decide what information the hidden state should carry.

LSTM

- There are several Python packages implementing deep learning methods, including `tensorflow` and `pytorch`. Version 2 of `tensorflow` appears to emerge as the industry standard.
- The book [3] contains code samples in including `tensorflow`.

References

[1] Aggarwal, C. C.:*Neural Networks and Deep Learning*, Springer (2018).

[2] Pincus, A.: Approximation theory of the MLP model in neural networks, *Acta Numerica*, **8**, 143 - 195 (1999).

[3] Geron, A.: *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, O'Reilly (2019).

[4] Goodfellow, I., Bengio, Y., and Courville, A.: *Deep Learning*, MIT Press (2016).

[5] Hochreiter, S., and Schmidhuber, J.: Long short-term memory, *Neural Computation*, **9**, 1735 – 1780 (1997).