Lecture Notes

Institute for Machine Learning, Johannes Kepler University Linz

JOHANNES KEPLER UNIVERSITY LINZ

LSTM and Recurrent Nets

Winter Semester 2019

by Sepp Hochreiter and Thomas Adler

Institute for Machine Learning
Johannes Kepler University Linz
A-4040 Linz, Austria

Tel. +43 732 2468 4530
https://www.jku.at/iml

Chapter 1

Simple Recurrent Networks

Recurrent neural networks (RNNs) are a large class of models, that can be seen as *feedforward neural networks*, augmented by connections that link between multiple forward passes. This means that a new activation depends on both, the current input variables and previous or old activations. Usually this additional dimension is thought of as time, although there is no necessary correspondence to the physical concept of *time*. The foundational research for RNNs was conducted in the 1980s and 1990s, and pervade many of today's application (e.g. Apple, 2018; Naik et al., 2018; OpenAI, 2019; DeepMind, 2019). The goal of these lecture notes is to offer a comprehensive view of their functionality as well showing their historical development. Both of which should ultimately provide a deeper understanding of the reasons for their success.

A feed-forward network can be considered as a function $\hat{\boldsymbol{y}} = g(\boldsymbol{x}; \boldsymbol{w})$ that maps an input vector $\boldsymbol{x}$ to an output (or prediction) vector $\hat{\boldsymbol{y}}$ using network parameters $\boldsymbol{w}$. That is, the forward pass activates the network depending on the input variables only and produces output values. Accordingly, RNNs map an input sequence $(\boldsymbol{x}(t))_{t=1}^{T}$ to an output sequence $(\hat{\boldsymbol{y}}(t))_{t=1}^{T}$ by

$$\hat{\boldsymbol{y}}(t) \;=\; g\left(\boldsymbol{a}(0), \boldsymbol{x}(1), \ldots, \boldsymbol{x}(t); \boldsymbol{w}\right) \;. \tag{1.1}$$

Here, the vector $\boldsymbol{a}(0)$ represents the initial recurrent activations, discussed later. We use $\hat{\boldsymbol{y}}(t)$ as network output because $\boldsymbol{y}(t)$ will later be used for labels and the network outputs can be interpreted as estimates for the labels, which is indicated by the hat symbol.

Feed-forward networks can in principle also be used for time series prediction by combining the vectors of the sequence $(\boldsymbol{x}(t))_{t=1}^{T}$ into a single vector of size $T \dim(\boldsymbol{x})$ and use it as input variable. However, this requires T to be constant, i.e. the network can only process sequences of length T, otherwise the architecture of the network must be altered and one has to train from scratch. Alternatively, the feed-forward net could operate on fixed-size contiguous subsequences of the input sequence. This is called a *sliding window* approach, which has the drawback that the network cannot see information outside of the current window and the number of parameters grows with the window size.

By contrast, recurrent networks can process arbitrarily long sequences with a constant number of parameters and can memorize information over long distances. For these reasons, they are a very natural and elegant solution to neural sequence processing tasks. More importantly, RNNs allow for *temporal generalization* in contrast to the sliding window approach. Temporal generalization means that learning to store important information at a certain time step can be generalized to store this information also if it appears at time steps that are never seen during training. For

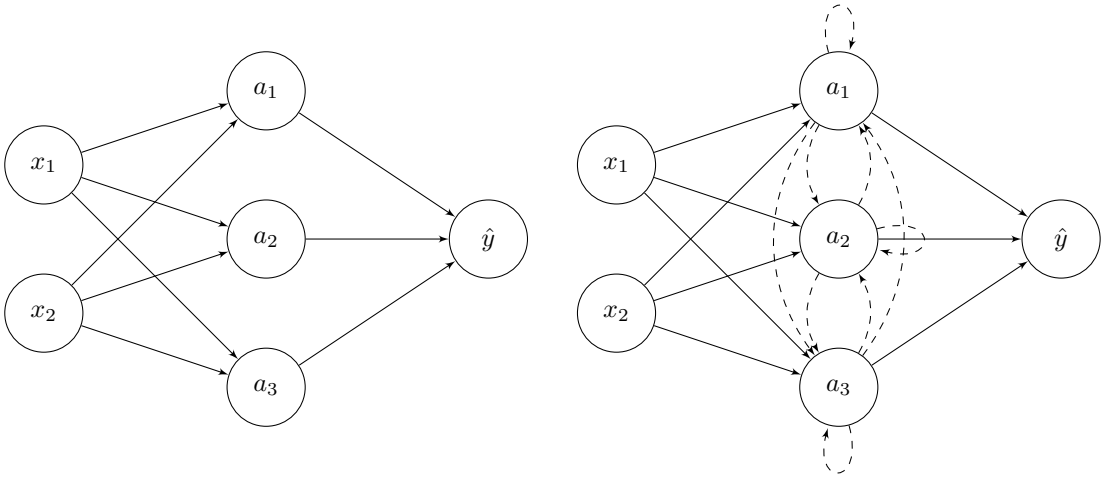

Figure 1.1: Fully connected vs. recurrent network. The *left* graph shows a simple feedforward network with input layer, hidden layer, and output layer. The *right* graph a recurrent network. It has the same basic architecture, but with recurrent connections (loops) in the hidden layer. The dashed lines indicate time lag, i.e. the transformation takes values at time $t-1$ and feeds them back into the network at time t. As you can see, the loop connections make the difference between these two architectures.

example an important input that is seen in the training set at time step 5,7,8,9 but never at time step 6, is generalized by RNNs to time step 6 while feed-forward networks fail.

It has been shown that RNNs are Turing-complete (Siegelmann and Sontag, 1991; Sun et al., 1991; Siegelmann, 1995). Informally, this means that every computer program can be represented by an RNN, which indicates their high potential. However, while theoretically important, these results make no statement about how to obtain a corresponding RNN representation.

1.1 Jordan network

One of the earliest recurrent neural architectures is the *Jordan network* (Jordan, 1986). It consists of a neural network with one hidden layer that feeds its outputs at time $t-1$ back as inputs at time t. The basic idea is to keep the last output as a form of context for processing the next input. In this fashion the Jordan network is able to *"remember"* and learn temporal patterns. The forward rule is

$$\begin{aligned} \boldsymbol{s}(t) &= \boldsymbol{W}^\top \boldsymbol{x}(t) + \boldsymbol{R}^\top \hat{\boldsymbol{y}}(t-1) \\ \boldsymbol{a}(t) &= f(\boldsymbol{s}(t)) \\ \hat{\boldsymbol{y}}(t) &= \varphi(\boldsymbol{V}^\top \boldsymbol{a}(t)), \end{aligned} \tag{1.2}$$

where $\boldsymbol{W} \in \mathbb{R}^{D\times I}, \boldsymbol{R} \in \mathbb{R}^{K\times I}, \boldsymbol{V} \in \mathbb{R}^{I\times K}$ are weight matrices, whose entries are collected in the parameter vector $\boldsymbol{w}$. The vector $\boldsymbol{s}(t) \in \mathbb{R}^I$ holds the pre-activations at time t and $\boldsymbol{a}(t) \in \mathbb{R}^I$ holds the hidden activations of the network at time t.

Note that all activations (including inputs and outputs) are time dependent but the weights $\boldsymbol{W}, \boldsymbol{R}, \boldsymbol{V}$ are not. We use the same weights in all time steps. Therefore, if we update a weight

we affect the system behavior at every time step. This concept is called *weight sharing* because all time steps share the same parameters. Due to weight sharing, the number of weights remains constantly independent from the length of the sequence without restricting the network's memory scope backwards in time.

The matrix $\boldsymbol{W}$ is the *input weight matrix*. It is a transformation between the input space $\mathbb{R}^D$ and the hidden space $\mathbb{R}^I$, i.e. it maps input features to hidden representations. The matrix $\boldsymbol{R}$ is the *recurrent weight matrix*. It holds the weights of the loop connections and determines the time-dependent behavior of the system. It maps hidden representations forward in time and empowers the network to "remember" things from the past. At every time step t the network has access to the outputs of the previous time step $t-1$. Finally, the matrix $\boldsymbol{V}$ is the *output weight matrix* which maps the network's hidden representations to the output space (or target space) $\mathbb{R}^K$. Figure 1.2 depicts the functional dependencies in this network.

Note: Bias units

For notational convenience, we neglect bias units when computing pre-activations. The reason for this is that bias units, while practically important, do not really change the math, which can be seen by a simple trick. Reconsider Equation (1.2). An actual implementation would use the form

$$\begin{aligned} \boldsymbol{s}(t) &= \boldsymbol{W}^\top \boldsymbol{x}(t) + \boldsymbol{R}^\top \hat{\boldsymbol{y}}(t-1) + \boldsymbol{b} \\ \boldsymbol{a}(t) &= f(\boldsymbol{s}(t)) \\ \hat{\boldsymbol{y}}(t) &= \varphi(\boldsymbol{V}^\top \boldsymbol{a}(t) + \boldsymbol{c}), \end{aligned} \tag{1.3}$$

where $\boldsymbol{b} \in \mathbb{R}^I$ and $\boldsymbol{c} \in \mathbb{R}^K$ are trainable weights, much like $\boldsymbol{W}, \boldsymbol{R}, \boldsymbol{V}$. Now we can extend the input vector $\boldsymbol{x}(t)$ by an additional dimension with a constant value of 1, i.e. $\bar{\boldsymbol{x}}(t) = (x_1, \ldots, x_D, 1)^\top \in \mathbb{R}^{D+1}$ and adjust $\boldsymbol{W}$ accordingly, such that $\bar{\boldsymbol{W}} \in \mathbb{R}^{(D+1)\times I}$, that is we have I additional entries. Collecting these entries in a vector $\boldsymbol{b}$, we have that $\bar{\boldsymbol{W}}^\top \bar{\boldsymbol{x}}(t) = \boldsymbol{W}^\top \boldsymbol{x}(t) + \boldsymbol{b}$. Of course, the same trick applies to $\boldsymbol{V}, \boldsymbol{a}(t), \boldsymbol{c}$.

The function f is called *activation function* or *non-linearity*. We usually define such functions as a scalar mapping $f : \mathbb{R} \to \mathbb{R}$. Whenever its argument is a vector, matrix, or tensor, we mean *pointwise* application. That is we apply the scalar mapping in each dimension of the input variable individually. This requires input and output dimensions to be equal and the distinct dimensions are independent from each other in the sense that changing the input value in one dimension does not affect the function value in any other dimension. Mathematically speaking, if we have $f : \mathbb{R}^I \to \mathbb{R}^I$, then

$$\left(\frac{\partial f(\boldsymbol{x})}{\partial x_i}\right)_{j \neq i} = 0. \tag{1.4}$$

That is, the gradient of f has a diagonal structure. A few examples for activation functions used in practice are the rectified linear unit (ReLU) $f(x) = \max(0, x)$, the logistic sigmoid $f(x) = (1 + e^{-x})^{-1}$, or the hyperbolic tangent

$$f(x) = (e^x - e^{-x})(e^x + e^{-x})^{-1}. \tag{1.5}$$

The function φ is the output activation function and mainly depends on the task at hand and not on the architecture of the network. In case of least-squares regression, we would usually

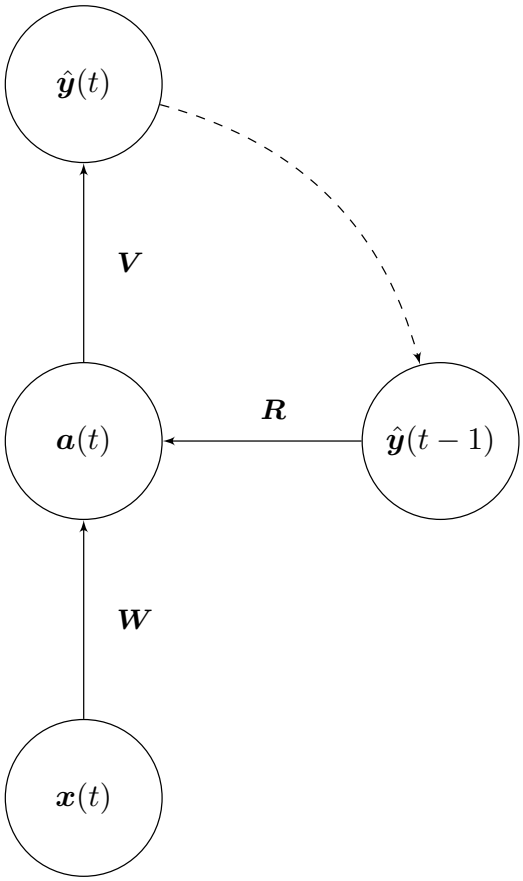

Figure 1.2: Jordan Network. The Jordan network "remembers" its previous outputs by feeding them back into the network at the current time step. The matrix $\boldsymbol{R}$ defines how the previous outputs are processed.

choose $\varphi(x) = x$ to be the identity function, i.e. we have linear output units. In case of binary classification, a natural choice would be the logistic sigmoid function $\varphi(x) = \sigma(x) = (1+e^{-x})^{-1}$ (in this case we prefer $\sigma(x)$ to denote the function) together with the cross-entropy loss as known from logistic regression. In case of classification with three or more classes, we use the softmax function

$$\varphi_i(\boldsymbol{x}) = \sigma_i(\boldsymbol{x}) = \frac{e^{x_i}}{\sum_{j=1}^{\dim \boldsymbol{x}} e^{x_j}} \tag{1.6}$$

usually also in combination with the cross entropy loss function.

Jordan (1986) proposed to train this network locally in time. That is at each time step t the network error is evaluated for this time step only and the weights are adjusted immediately. This means that the error signal is not propagated back in time. However, the architecture admits for other learning algorithms as well, some of which will be discussed later.

When training a Jordan network, there exists an interesting variant that is nowadays known as *teacher forcing*. During training, we have labels $\boldsymbol{y}(1{:}T)$ available. We can make use of them not only as target values but also as recurrent inputs in that we replace Equation (1.2) by

$$\begin{aligned} \boldsymbol{s}(t) &= \boldsymbol{W}^\top \boldsymbol{x}(t) + \boldsymbol{R}^\top \boldsymbol{y}(t-1) \\ \boldsymbol{a}(t) &= f(\boldsymbol{s}(t)) \\ \hat{\boldsymbol{y}}(t) &= \varphi(\boldsymbol{V}^\top \boldsymbol{a}(t)) \,, \end{aligned} \tag{1.7}$$

the only difference being the use of $\boldsymbol{y}(t-1)$ instead of $\hat{\boldsymbol{y}}(t-1)$ to feed the recurrent connections. Of course, this is only possible during training where we have labels $\boldsymbol{y}(1{:}T)$ available. In inference mode, the network has do deal with its own predictions.

1.2 Elman network

A modification of the Jordan network is the *Elman network* introduced by Elman (1988). Often when the term *simple recurrent neural network* is used it actually refers to the Elman network. The network links the recurrent connections to the hidden units instead of the output units. The network remembers internal hidden activations instead of output activations. This is a more efficient representation because the network does not need to encode and decode its memories at every time step. The forward pass is

$$\begin{aligned} \boldsymbol{s}(t) &= \boldsymbol{W}^\top \boldsymbol{x}(t) + \boldsymbol{a}(t-1) \\ \boldsymbol{a}(t) &= f(\boldsymbol{s}(t)) \\ \hat{\boldsymbol{y}}(t) &= \varphi(\boldsymbol{V}^\top \boldsymbol{a}(t)), \end{aligned} \tag{1.8}$$

where $\boldsymbol{W} \in \mathbb{R}^{D\times I}, \boldsymbol{V} \in \mathbb{R}^{I\times K}$ and $\dim \boldsymbol{w} = I(D+K)$. Note that we can describe the recurrent connections in this setting equivalently using the identity matrix $\boldsymbol{I}_I$. This means that the loop connections only link the hidden units to themselves with a constant weight of 1 but not to neighboring units. Figure 1.3 shows the essence of Elman's architecture.

The forward pass starts with an input signal $\boldsymbol{x}(t)$, which is fed to the network. Together with the weight matrix $\boldsymbol{W}$ and the old hidden activations $\boldsymbol{a}(t-1)$, this activates the new hidden activations $\boldsymbol{a}(t)$. A final transformation governed by $\boldsymbol{V}$ activates the output units $\hat{\boldsymbol{y}}(t)$. If at this

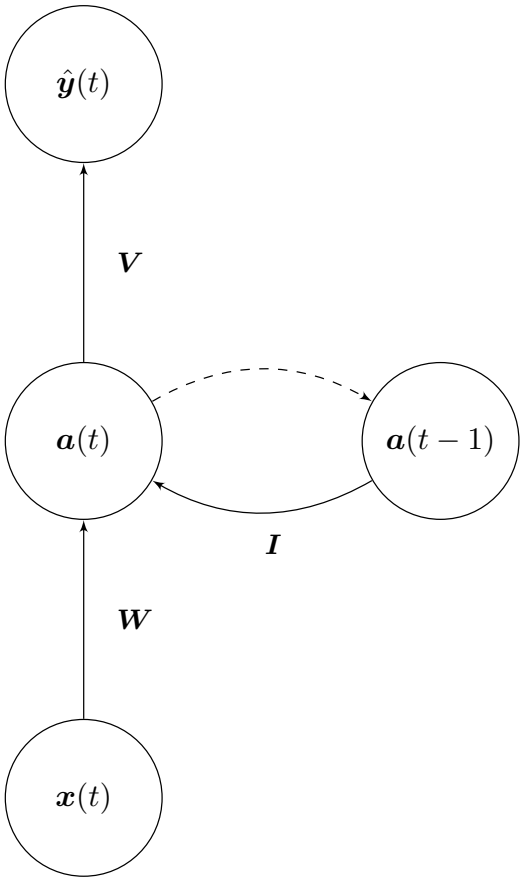

Figure 1.3: Elman network. The Elman network is similar to the Jordan network but instead of looping back the previous outputs the Elman network "remembers" the previous hidden activations $\boldsymbol{a}(t-1)$. The recurrent connections are one-to-one, i.e. can be formally described by an $I \times I$ identity matrix, and are not subject to weight updates, i.e. they are constant.

time t a label $\boldsymbol{y}(t)$ is available, a loss function may compare $\hat{\boldsymbol{y}}(t)$ to $\boldsymbol{y}(t)$ and initiate a learning signal. This is where the backward pass starts, which traverses backwards through the network to compute the gradients of the loss function with respect to the network parameters. These gradients are then used for updating the network parameters into a direction where the loss becomes smaller (or more precisely, into the direction of *steepest descent* of the loss). Elman (1988) proposed to exclude the recurrent connections from training. As we will see later, when backpropagation through time is used as learning rule, then this choice of constant recurrent weights provides the learning algorithm with some stability (cf. *vanishing gradient*). However, similar to Jordan (1986), Elman (1988) trained his network by using backpropagation on each time step individually. That is, the recurrent connection is ignored during the weight updates, so that the time dependency is not captured in the updating procedure.

Note: Training Elman networks

Cleeremans et al. (1989) refer to updating strategy of Jordan (1986) and Elman (1988) as *completely local in time* and some publications refer to it as *Elman Training Procedure* or *Elmans Training Procedure* (e.g. Hochreiter, 2001)

1.3 Fully recurrent network

We will now modify the Elman network in that we loosen the choice of recurrent parameters. Instead of using constant identity, we will now allow for any choice of recurrent parameters $\boldsymbol{R} \in \mathbb{R}^{I \times I}$. We will refer to the resulting model as the *fully recurrent network*. The forward pass becomes

$$\begin{aligned} \boldsymbol{s}(t) &= \boldsymbol{W}^\top \boldsymbol{x}(t) + \boldsymbol{R}^\top \boldsymbol{a}(t-1) \\ \boldsymbol{a}(t) &= f(\boldsymbol{s}(t)) \\ \hat{\boldsymbol{y}}(t) &= \varphi(\boldsymbol{V}^\top \boldsymbol{a}(t)), \end{aligned} \tag{1.9}$$

where $\boldsymbol{W} \in \mathbb{R}^{D \times I}$, $\boldsymbol{R} \in \mathbb{R}^{I \times I}$, $\boldsymbol{V} \in \mathbb{R}^{I \times K}$ and $\dim \boldsymbol{w} = I(D + I + K)$. In contrast to the Elman network, we will now treat the recurrent weights $\boldsymbol{R}$ as trainable, i.e. we will apply weight updates to $\boldsymbol{R}$. As opposed to Elman networks, a hidden unit may now not only depend on a former version of itself but also of all neighboring neurons. The architecture is depicted in Figure 1.4.

Equation (1.9) raises an issue when we try to process the very first element $\boldsymbol{x}(1)$ of an input sequence. It seems we have to know the values of the hidden activations at time $t = 0$. Therefore, we have to provide an initial activation $\boldsymbol{a}(0)$ as indicated in equation (1.1). Often, $\boldsymbol{a}(0) = \boldsymbol{0}$ is a reasonable choice as this corresponds to a "clean" memory.

Figure 1.5 shows how the RNN processes an input sequence. It can be imagined as sliding the network over the sequence because at each time step we use the same set of weights, which define the network behavior. At each time step t it reads an input element $\boldsymbol{x}(t)$ while the loop connections provide information about the past, which is stored in the previous hidden activations $\boldsymbol{a}(t-1)$. This leads to a new hidden activation $\boldsymbol{a}(t)$ which is then transformed to an output activation $\hat{\boldsymbol{y}}(t)$. A loss function compares this output activation to the corresponding target $\boldsymbol{y}(t)$.

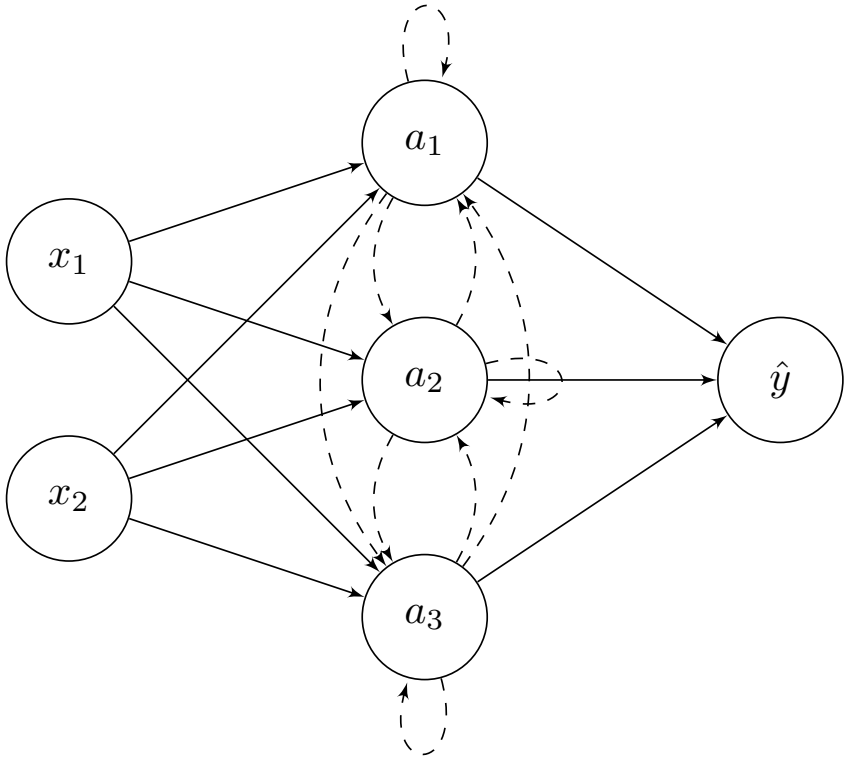

Figure 1.4: Fully recurrent network. The recurrent hidden layer is fully connected, which means that all units are interconnected, so that the hidden units are able to store information (i.e. information from previous inputs is kept in the hidden units). The recurrent connections have time lag, which is indicated by dashed lines.

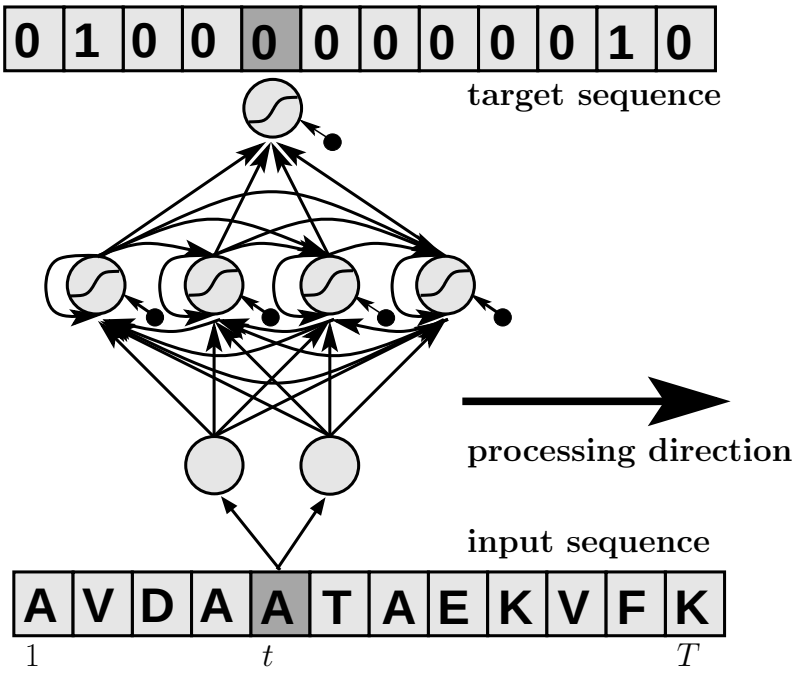

Figure 1.5: Processing of a sequence with an RNN. Here, the sequence starts at time step 1, the current timestep is indicated by t, and the end of the sequence is T. At each time step the current input element is fed to the recurrent network. The *weight sharing* can be imagined as sliding the network over the input sequence.

1.4 ARMA

The term *ARMA* is an acrony for the **a**uto**r**egressive-**m**oving-**a**verage (model). In the following, we will consider a one dimensional setting, i.e. the inputs are a sequence of a single feature $x(t)$. An autoregressive model of order P tries to forecast one time step ahead of this sequence while looking at the P most recent time steps, i.e.

$$x(t) = c + \sum_{p=1}^{P} w_p x(t-p) + \varepsilon(t), \tag{1.10}$$

where w_p and c are to-be-estimated model parameters. The autoregression task may be thought of as a prediction task where inputs and label stem from the same sequence, i.e. given some history $x(1), \ldots, x(t)$, predict $x(t+1)$.

A moving-average model tries to make a forecast by only looking at the noise terms. It has the form

$$x(t) = \mu + \sum_{q=1}^{Q} v_q \varepsilon(t-q) + \varepsilon(t), \tag{1.11}$$

where μ is the expected value of $x(t)$. It is meant to predict unexpected shocks in the sequence.

The ARMA model was considered by Whittle (1951) and popularized by Box and Jenkins (1970). It combines an autoregressive-model of order P and a moving-average model of order Q. As such, it can be considered as a linear recurrent network, which has the form

$$x(t) = c + \sum_{p=1}^{P} w_p x(t-p) + \sum_{q=1}^{Q} v_q \varepsilon(t-q) + \varepsilon(t), \tag{1.12}$$

where P is the order of the autoregressive term, and Q is the order of the moving-average term. The variables $\varepsilon(t)$ represent noise and $x(t)$ is a given sequence. The weights w_p and v_q are chosen such that the noise becomes minimal.

1.5 NARX recurrent neural networks

Non-linear autoregressive exogenous models (NARX) are time series models of the form

$$\hat{\boldsymbol{y}}(t) = \boldsymbol{g}\big(\hat{\boldsymbol{y}}(t-1), \ldots, \hat{\boldsymbol{y}}(t-T_y), \boldsymbol{x}(t), \ldots, \boldsymbol{x}(t-T_x)\big), \tag{1.13}$$

where $\hat{\boldsymbol{y}}(t-1), \ldots, \hat{\boldsymbol{y}}(t-T_y)$ constitutes the autoregressive, and $\boldsymbol{x}(t), \ldots, \boldsymbol{x}(t-T_x)$ the e**x**ogenous part. If $\boldsymbol{g}$ is realized by a neural network, the model is called NARX recurrent neural network. The recurrence arises from the model outputs $\hat{\boldsymbol{y}}(t-1), \ldots, \hat{\boldsymbol{y}}(t-T_y)$ being fed back to the system as inputs. The constants T_y and T_x can be seen as window sizes, which delimit the context that can be seen by the system in terms of output and input variables, respectively.

The Jordan network (see Equation 1.2) can be seen as a trivial instance of a NARX recurrent net with $T_y = 1$ and $T_x = 0$. Increasing T_y creates shortcuts to network outputs further in the past. Lin et al. (1996) argued that these shortcuts stabilize learning and dependencies between events

not more than T_y time steps apart can be captured by the gradient without decay. However, this comes at the cost of an increasing parameter set, i.e. the number of parameters grows with T_y.

Thus, NARX recurrent nets are able to avoid the vanishing gradient problem for a finite number of time steps but they do not solve the problem of long-term dependencies rigorously.

1.6 Time-delay neural networks

Time-delay neural networks (TDNNs) Waibel et al. (1989) were originally designed for phoneme recognition, i.e. part-of-speech classification.

Every connection w_{ij} from one unit i to another unit j has $N + 1$ different values w_{ijn} for the N delays $0, D_1, \ldots, D_n, \ldots, D_N$. The value $a_i(t)w_{ijn}$ is added to the pre-activation of unit j at time $t + D_n$. The activation $a_i(t)$ has synaptic strength w_{ijn} to unit j with delay D_n. For the special case that $D_n = n$, we have a 1-D convolutional network. On the other hand each 1-D convolutional network which has window size larger than or equal to $\max_{ij} D_N$ can represent the TDNN with delays D_n. The time delays can be learned, e.g. by using a softmax over all delays between 1 and $D_{\max}$. The softmax converges during learning to a one-hot encoding, therefore, selecting one specific delay. However learning the delay is as computational intensive as a 1-D convolutional network.

Again, the problem of long-term dependencies does not occur within the window size but the number of weights grows with the window size.

Bibliography

Apple, F. N. L. P. T. (2018). Can global semantic context improve neural language models? https://machinelearning.apple.com/2018/09/27/can-global-semantic-context-improve-neural-language-models.html. Accessed: 2019-10-10.

Box, G. E. and Jenkins, G. (1970). *Time series analysis: forecasting and control*. San Francisco: Holden-Day.

Cleeremans, A., Servan-Schreiber, D., and McClelland, J. L. (1989). Finite-state automata and simple recurrent networks. *Neural Computation*, 1:372–381.

DeepMind (2019). Alphastar: Mastering the real-time strategy game starcraft ii. https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii. Accessed: 2019-10-10.

Elman, J. L. (1988). Finding structure in time. Technical Report CRL 8801, Center for Research in Language, University of California, San Diego.

Hochreiter, S. (2001). *Generalisierung bei Neuronalen Netzen geringer Komplexität*. infix, DISKI 202, ISBN 3-89838-202-8. Akademische Verlagsgesellschaft Aka GmbH, Berlin.

Jordan, M. I. (1986). Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of Ninth Annual Conference of the Cognitive Science Society, Amherst*, pages 531–546.

Lin, T., Horne, B. G., Tino, P., and Giles, C. L. (1996). Learning long-term dependencies in NARX recurrent neural networks. *IEEE Transactions on Neural Networks*, 7(6):1329–1338.

Naik, C., Gupta, A., Ge, H., Mathias, L., and Sarikaya, R. (2018). Contextual slot carryover for disparate schemas. *arXiv preprint arXiv:1806.01773*.

OpenAI (2019). Openai five. https://openai.com/five/. Accessed: 2019-10-10.

Siegelmann, H. (1995). Computation beyond the turing limit. *Science*, 268:545–548.

Siegelmann, H. and Sontag, E. (1991). Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80.

Sun, G., Chen, H., Lee, Y., and Giles, C. (1991). Turing equivalence of neural networks with second order connection weights. In *IEEE INNS International Joint Conference on Neural Networks*, volume II, pages 357–362, Piscataway, NJ. IEEE Press.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., and Lang, K. (1989). Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics Speech and Signal Processing*.

Whittle, P. (1951). Hypothesis testing in time series analysis (uppsala, sweden: Almqvist and wicksell).