Lecture Notes

Institute for Machine Learning, Johannes Kepler University Linz

JOHANNES KEPLER UNIVERSITY LINZ

LSTM and Recurrent Neural Nets

Winter Semester 2021

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

Contents

List of Figures

List of Tables

Chapter -1

Topics

-1.1 Introduction

What is an RNN in contrast to feedforward neural networks.

-1.2 Architectures

- Elman nets
- Jordan nets
- NARX nets
- time-delay neural networks (Alex Waibel)
- LSTM nets
- GRU nets
- Hinton "A Simple Way to Initialize Recurrent Networks of Rectified Linear Units", SELUs
- other architectures:
    - attractor networks (Almeida 1987, Pineda 1988)
    - Hopfield networks (Hopfield 1982, Little 1974)
    - Boltzmann machines (Hinton & Sejnowski 1985)

-1.3 Learning methods

- Real-Time Recurrent Learning (RTRL)
- Backpropagation through time (BPTT)
- truncated BPTT
- focused backpropagation (Mike Mozer)

-1.4 Regularization methods

- weight decay
- early stopping
- dropout
- keeping state changes small (David Krueger, Zoneout)

-1.5 Vanishing gradient problem for RNNs

-1.6 Special Architectures

- time-delay neural networks (Alex Waibel)
- echo state networks
- bidirectional RNNs
- recursive networks

-1.6.1 LSTM-based Architectures

- ConvLSTM
- Grid-LSTM
- Pyramid-LSTM
- Neural Turing Machine
- Sequence-to-Sequence Learning
- Tree-LSTM? `https://arxiv.org/abs/1503.00075`

-1.7 Attention mechanisms

- Attention
- Transformer Networks and BERT
- Tinker

-1.8 Meta-Learning with RNNs

- Meta-Learning with LSTM

-1.9 Reinforcment Learning with RNNs

- RUDDER
- LSTM agents
- recurrent actor-critic
- Bram Bakker's architecture

-1.10 RNNs for credit assignment

- uniform credit assignment
- attention mechanisms for credit assignment

-1.11 Applications

- RNN in speech
- RNN for text analysis
- RNN for translation
- RNN for time series prediction
- RNN in finance
- RNN in hydrology

-1.12 RNNs win challenges

- speech,
- language
- text
- translation

-1.13 Highlights

- OpenAI uses LSTM Agents for Dota2
- DeepMind uses LSTM Agents for StarCraftII

-1.14 Notation

Chapter 0

Notation

0.1 Scalars, vectors, and matrices

For *scalars*, we use simple lower-case, italic letters, such as x, y, or a. We use bold, lower-case symbols for denoting *vectors*, such as $\boldsymbol{x}$ and $\boldsymbol{w}$ — these represent *column vectors*. Finally, we use bold, capital letters to denote *matrices*, for example $\boldsymbol{X}$, $\boldsymbol{W}$ or $\boldsymbol{Y}$.

0.2 Sizes and dimensions

- N: number of training examples, i.e. objects or samples, in the *training data set*. Running index: n.
- D: number of input units which is also the number of features that a sample has. Running index: d
- M: number of test or validation examples, i.e. objects or samples, in the *test data set*. I think we should keep m for a second sample.. eg for SVMs. We only need the test set at a single specific section. Running index: m with $N + 1 \leqslant m \leqslant N + M$.
- K: number of output units. Running index: k.
- L: number of layers in a network. Running index: l.
- I: input dimension of a general neural network layer. Running index: i.
- J: output dimension of a general neural network layer. Running index: j.

0.3 Objects

A superscript ^{n} will denote the n-the training example.

- $\boldsymbol{x}^n \in \mathbb{R}^D$ or $\boldsymbol{x} \in \mathbb{R}^D$: the n-th input data point or a general data point, respectively. A column vector. Sometimes also used for the inputs of a particular neural network layer, then the dimensions could be $\boldsymbol{x} \in \mathbb{R}^J$.
- $y^n \in \mathbb{R}$ or $y \in \mathbb{R}$: a scalar label for the n-th data point or a general label y, respectively.

- $\boldsymbol{y}^n \in \mathbb{R}^K$ or $\boldsymbol{y} \in \mathbb{R}^K$: For multi-class of multi-task problems, this is a *label vector* for the n-th data point or a general label vector, respectively. A column vector.
- $\boldsymbol{w} \in \mathbb{R}^D$: a *weight* or *parameter* vector of a simple machine learning method, such as linear regression. A column vector.
- $\boldsymbol{W} \in \mathbb{R}^{I \times J}$: a *weight* or *parameter* matrix of a learning method mapping from an input space with dimension J to an output space with dimension I.
- $\boldsymbol{b} \in \mathbb{R}^I$: a bias vector.
- $\hat{y} \in \mathbb{R}$ or $\hat{\boldsymbol{y}} \in \mathbb{R}^K$: the predicted scalar label or – for a multi-class problem – the predicted label vector, respectively.
- $p \in \mathbb{R}$ or $\boldsymbol{p} \in \mathbb{R}^K$: same as above if outputs can be interpreted probabilistically, the predicted scalar label or – for a multi-class problem – the predicted label vector.
- $\boldsymbol{\theta}$: a set of parameters of a probabilistic model.
- $\boldsymbol{a} \in \mathbb{R}^I$: *activations* of a neural network.
- $\boldsymbol{s} \in \mathbb{R}^I$: *pre-activations*, also called *netI*, of a neural network.

Note that there can be two kinds of subscripts:

- $\boldsymbol{x}^n$: the n-th data point. This represents a *column vector*.
- x_d: the d-the component of the vector $\boldsymbol{x} = (x_1, \ldots, x_d, \ldots, x_D)$. This represents a *scalar*.

In rare cases, to denote the d-th component of the n-th data point, we will use the notation $x_{n,d}$ (see also "Stacked data and labels" below).

0.4 Multiple layers

For a neural network with multiple layers, the following notations are used:

- $n_h^{[l]}$: number of hidden units of the l-th layer. Thus, $D = n_h^{[0]}$ and $K = n_h^{[L+1]}$.
- $\boldsymbol{a}^{[l]} \in \mathbb{R}^{n_h^{[l]}}$: *activations* of a neural network in the l-th layer.
- $\boldsymbol{s}^{[l]} \in \mathbb{R}^{n_h^{[l]}}$: *pre-activations*, also called *netI*, of a neural network in the l-th layer.
- $\boldsymbol{W}^{[l]} \in \mathbb{R}^{n_h^{[l-1]} \times n_h^{[l]}}$: the weight matrix connecting the $(l-1)$-th layer with the l-th layer.
- $\boldsymbol{b}^{[l]} \in \mathbb{R}^{n_h^{[l]}}$: the bias vector in the l-th layer.

0.5 Stacked data and labels

At some points, the input data points $\boldsymbol{x}^1, \ldots, \boldsymbol{x}^n$ are stacked together to form a data matrix $\boldsymbol{X}$, which contains the data points $\boldsymbol{x}^n$ as *rows*. Scalar labels $y^1, \ldots, y^N$ are stacked together to form a vector $\boldsymbol{y}$, or vectorial labels $y^1, \ldots, y^N$ are stacked together to form a *label matrix* $\boldsymbol{Y}$, which contains the labels as *rows*.

- $\boldsymbol{X} \in \mathbb{R}^{N \times D}$: input data matrix. The rows represent objects, i.e. samples. We assume that objects, i.e. samples, are represented or described by *feature vectors* $\boldsymbol{x}^n$.
- $\boldsymbol{y} \in \mathbb{R}^N$: the scalar labels of all samples stacked to a column vector.
- $\boldsymbol{Y} \in \mathbb{R}^{N \times K}$: for multi-class or multi-task problems, stacked labels yield a *label matrix*.

Question to Sepp: Should we use Nabla operator $\nabla_{\boldsymbol{w}}$ or partial operators $\frac{\partial}{\partial \boldsymbol{w}}$?

Question to Sepp: Should we use $R_{\text{emp}}(\boldsymbol{y}, \boldsymbol{X}, \boldsymbol{w})$ for empirical error or something shorter like $E_D(\boldsymbol{w})$?? Should we drop dependencies on $\boldsymbol{X}$ and $\boldsymbol{y}$ for convenience? The error of a regularized method could be $E = \lambda E_D + (1 - \lambda) E_W$ ... we could also solve this by macros for now.

0.6 Common transformations

The parameters of a single neuron, so-called weights, are written as a vector $\boldsymbol{w} = (w_1, \ldots, w_d)^T \in \mathbb{R}^d$. We will repeatedly use *affine transformations*:

$$\boldsymbol{w}^T \boldsymbol{x} + b = w_1 x_1 + \ldots w_D x_D + b. \tag{1}$$

In neural networks, affine transformations into a multi-dimensional space are frequently used:

$$\boldsymbol{s} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}. \tag{2}$$

Typically, a *dummy-one* variable $x_0 = 1$ is assumed in order to keep the notation uncluttered: in this case, we can simply write $\boldsymbol{w}^T \boldsymbol{x}$ or $\boldsymbol{W}\boldsymbol{x}$, which implicitly includes biases.

Furthermore, a non-linear function $f : \mathbb{R} \to \mathbb{R}$, a so-called *activation function*, is typically applied element-wise to the resulting vectors:

$$\boldsymbol{a} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}). \tag{3}$$

Occasionally, expressions such as

$$\boldsymbol{w}^T \boldsymbol{w} = \sum_{i=1}^{I} w_i^2 = \|\boldsymbol{w}\|^2, \tag{4}$$

where $\|\boldsymbol{w}\|^2$ is the squared 2-norm of $\boldsymbol{w}$, are used – typically in context with regularization techniques.

0.7 Functions

- $f : \mathbb{R} \to \mathbb{R}$: a non-linear *activation function* that is applied element-wise to a vector or matrix.
- $g(\boldsymbol{x}; \boldsymbol{w})$: a machine learning model with input $\boldsymbol{x}$ and parameters $\boldsymbol{w}$.
- $p(\boldsymbol{x}; \boldsymbol{\theta})$: a probabilistic model with data point $\boldsymbol{x}$ and set of parameters $\boldsymbol{\theta}$.
- $L(y, g(\boldsymbol{x}; \boldsymbol{w}))$ or $L(y, \hat{y})$: a loss function L.
- $R_{\text{emp}}(\boldsymbol{y}, \boldsymbol{X}, \boldsymbol{w})$: an empirical error or risk function. Typically, the empirical error is an average of the loss function for a single training data point. For neural networks, this serves as a *cost function* that is minimized.

Chapter 1

Simple Recurrent Networks

Insert some paragraphs of RNN history covering Hopfield, Pineda, Almeida, Hinton.

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 feedforward 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.

feedforward 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 feedforward 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

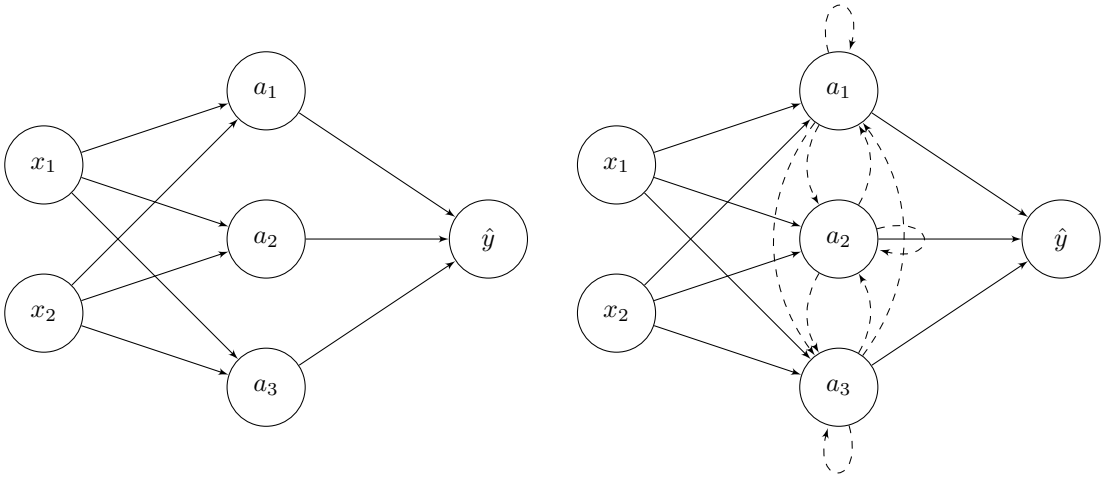


Figure 1.1: Fully connected vs. recurrent network. The *left* graph shows a simple feedforward network with input layer, hidden layer, and output layer. The *right* graph a recurrent network. It has the same basic architecture, but with recurrent connections (loops) in the hidden layer. The dashed lines 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.

to store this information also if it appears at time steps that are never seen during training. For 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 feedforward 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

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

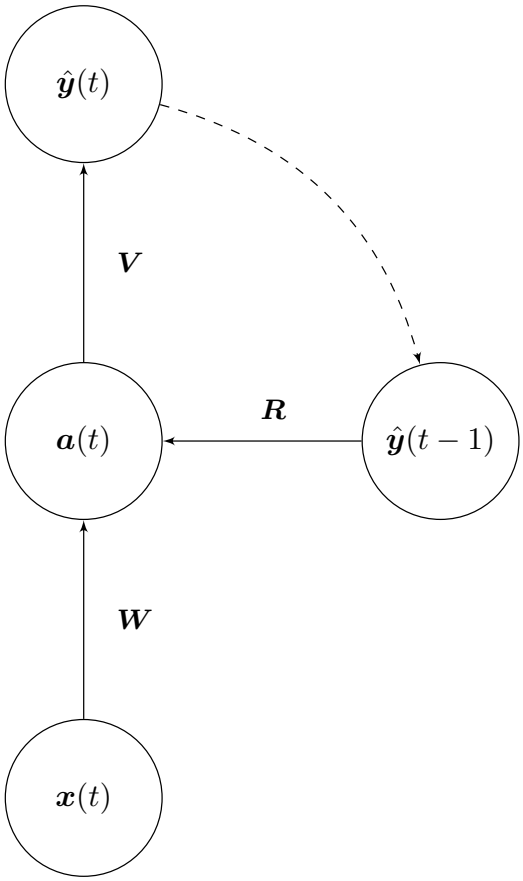


Figure 1.2: Jordan Network. The Jordan network "remembers" its previous outputs by feeding them back into the network at the current time step. The matrix $\boldsymbol{R}$ defines how the previous outputs are processed.

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 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.

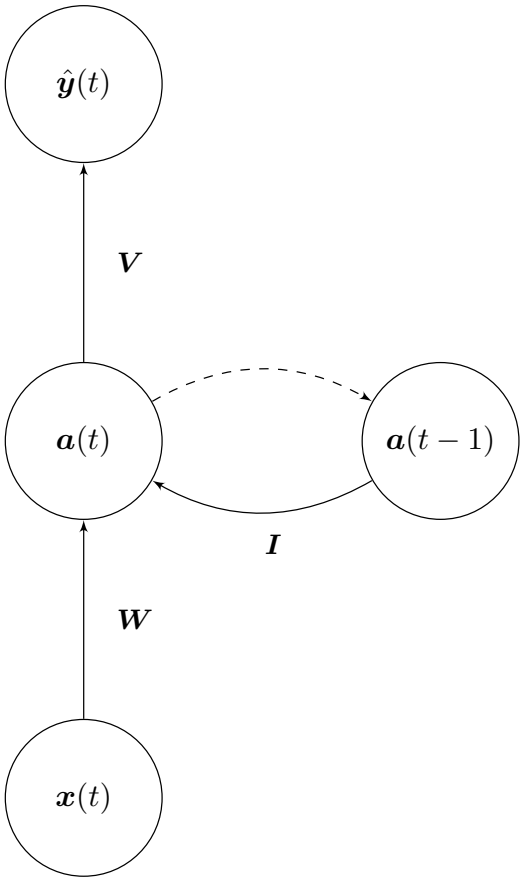


Figure 1.3: Elman network. The Elman network is similar to the Jordan network but instead of looping back the previous outputs the Elman network "remembers" the previous hidden activations $\boldsymbol{a}(t-1)$. The recurrent connections are one-to-one, i.e. can be formally described by an $I \times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

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 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 $\boldsymbol{a}(t-1)$.

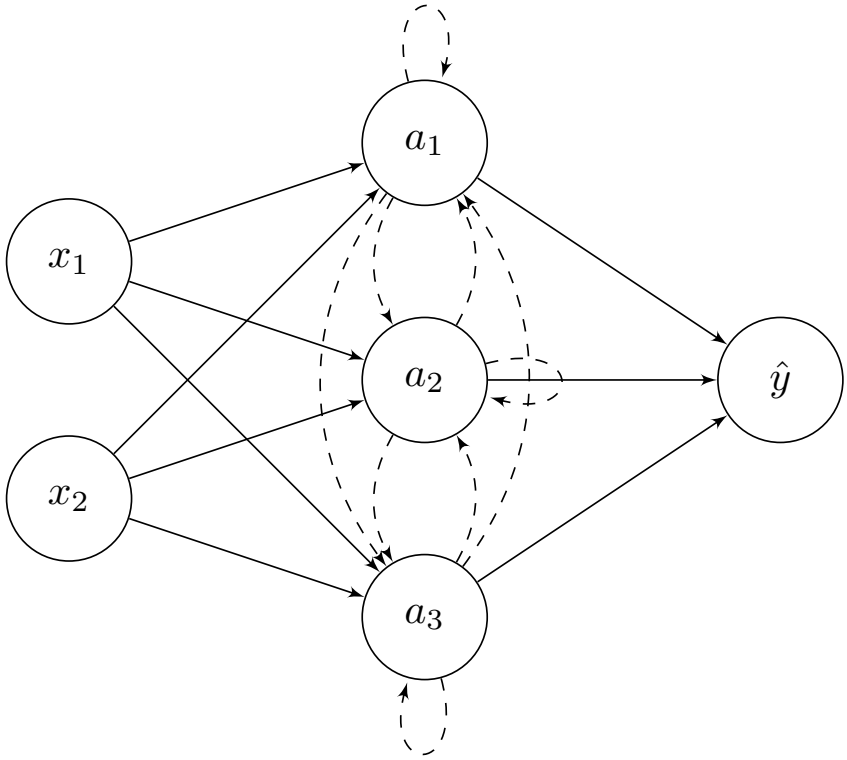


Figure 1.4: Fully recurrent network. The recurrent hidden layer is fully connected, which means that all units are interconnected, so that the hidden units are able to store information (i.e. information from previous inputs is kept in the hidden units). The recurrent connections have time lag, which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

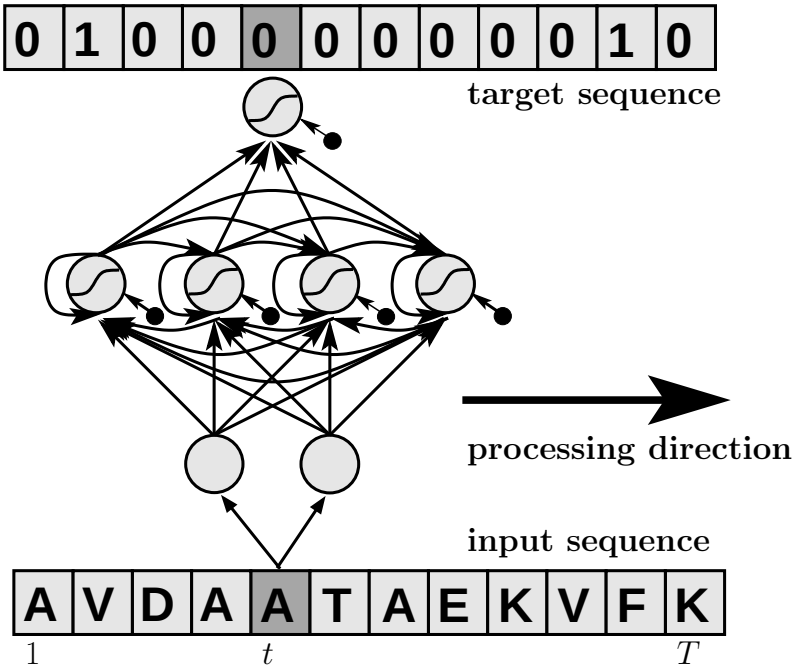


Figure 1.5: Processing of a sequence with an RNN. Here, the sequence starts at time step 1, the current timestep is indicated by t, and the end of the sequence is T. At each time step the current input element is fed to the recurrent network. The *weight sharing* can be imagined as sliding the network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

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 $\boldsymbol{y}(t)$.

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.

Chapter 2

Learning Algorithms for RNNs

2.1 Empirical risk minimization via gradient descent

In the following, let $\boldsymbol{x}(1{:}T) = (\boldsymbol{x}(t))_{t=1}^{T}$ denote an input sequence with T elements. Given a data set consisting of N input sequences $\{\boldsymbol{x}(1{:}T)^{(n)}\}_{n=1}^{N}$ and N target sequences $\{\boldsymbol{y}(1{:}T)^{(n)}\}_{n=1}^{N}$, we are concerned with minimizing the empirical risk

$$\begin{aligned} R_{\text{emp}}\left(\left\{\boldsymbol{y}(1{:}T)^{(n)}, \hat{\boldsymbol{y}}(1{:}T)^{(n)}\right\}_{n=1}^{N}\right) &= \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T} L\left(\boldsymbol{y}(t)^{(n)}, \hat{\boldsymbol{y}}(t)^{(n)}\right) \\ &= \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T} L\left(\boldsymbol{y}(t)^{(n)}, g\left(\boldsymbol{a}(0), \boldsymbol{x}(1{:}t)^{(n)}; \boldsymbol{w}\right)\right), \end{aligned} \tag{2.1}$$

where $L : \mathbb{R}^K \times \mathbb{R}^K \to \mathbb{R}$ is the loss function that measures how bad a prediction $\hat{\boldsymbol{y}}(t)^{(n)}$ is compared to the correct value $\boldsymbol{y}(t)^{(n)}$.

We want to find a parameterization $\boldsymbol{w}^*$ that minimizes the risk, i.e. to find

$$\boldsymbol{w}^* = \operatorname*{argmin}_{\boldsymbol{w}} R_{\text{emp}}(g(\cdot; \boldsymbol{w})). \tag{2.2}$$

One popular algorithm to determine $\boldsymbol{w}^*$ is *gradient descent*. Gradient descent is based on the observation that the negative gradient $-\nabla_{\boldsymbol{w}} R(g(\cdot; \boldsymbol{w}))$ points into the direction of steepest descent, that is the direction in parameter space where risk decreases fastest. Hence, for all parameterizations $\boldsymbol{w}^{\text{old}}$ there exists an $\eta \geqslant 0$, which we use to choose

$$\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} - \eta \nabla_{\boldsymbol{w}} R_{\text{emp}}(g(\cdot; \boldsymbol{w}^{\text{old}})) \tag{2.3}$$

such that $R_{\text{emp}}(g(\cdot; \boldsymbol{w}^{\text{new}})) \leqslant R_{\text{emp}}(g(\cdot; \boldsymbol{w}^{\text{old}}))$. We iterate this procedure until the only possible choice is $\eta = 0$ and we have converged to a local minimum. The step size η is called the *learning rate*.

2.2 Backpropagation Through Time

Computing the gradient for a feedforward net is called *backpropagation*. For recurrent networks this seems to be more complicated due to loop connections and weight sharing. However, with a

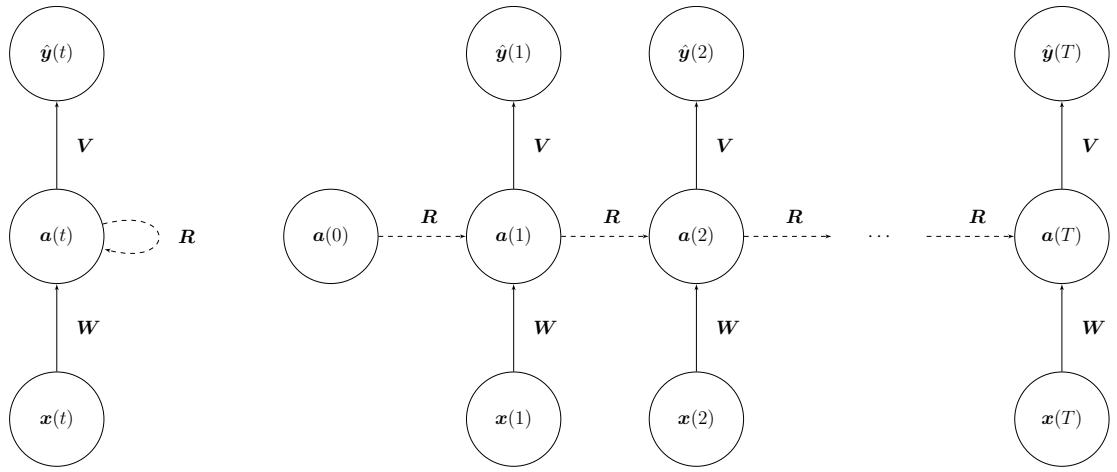


Figure 2.1: Left: A recurrent network. Right: The network from the left in feedforward formalism, where all units have a copy (*a clone*) for each time step. The network is said to be *unfolded* or *unrolled* in time.

slight change of view we can reduce this problem to that of computing the gradient of a feedforward network, namely by *unfolding/unrolling* the recurrent network *in time* (see Figure 2.1).

Backpropagation has two stages, the *forward pass* and the *backward pass*. The forward pass consists of activating the network with a new input and evaluating the resulting loss value. The backward pass (also called *delta propagation*) consists of computing the derivative of that loss w.r.t. the network parameters in a recursive fashion that starts at the output of the activated network and ends in the first layer.

It turns out that we can do the exact same thing for RNNs. The key to this insight is the observation that RNNs can be viewed as feedforward nets when we *unroll* their recurrent loops *in time*. For every time step t we copy the entire network and align this copy with the next time step $t + 1$. Then we break up the loop connections of the network at time t and link them to the corresponding units of the network copy at $t + 1$ (according to equation (1.9)). This results in a chain of T linked copies of the same network, which is depicted in Figure 2.1. The resulting architecture is mathematically equivalent to the looped version but has a feedforward structure on which we can apply the standard backpropagation algorithm. The only peculiarity is that the weights $\boldsymbol{W}, \boldsymbol{V}, \boldsymbol{R}$ occur T times in this structure but since this is just multiple usages of the same variables we have to store them only once. Compared to feedforward networks, backprop for RNNs not only propagates backwards through the layers but also through time, hence the name *backpropagation through time*.

Backpropagation through time (BPTT) is the standard method for training RNNs (Williams and Zipser, 1992; Werbos, 1988; Pearlmutter, 1989; Werbos, 1990). The shared weights need no special treatment. We can simply add the updates resulting from different time steps. This is analogous to adding updates resulting from different samples.

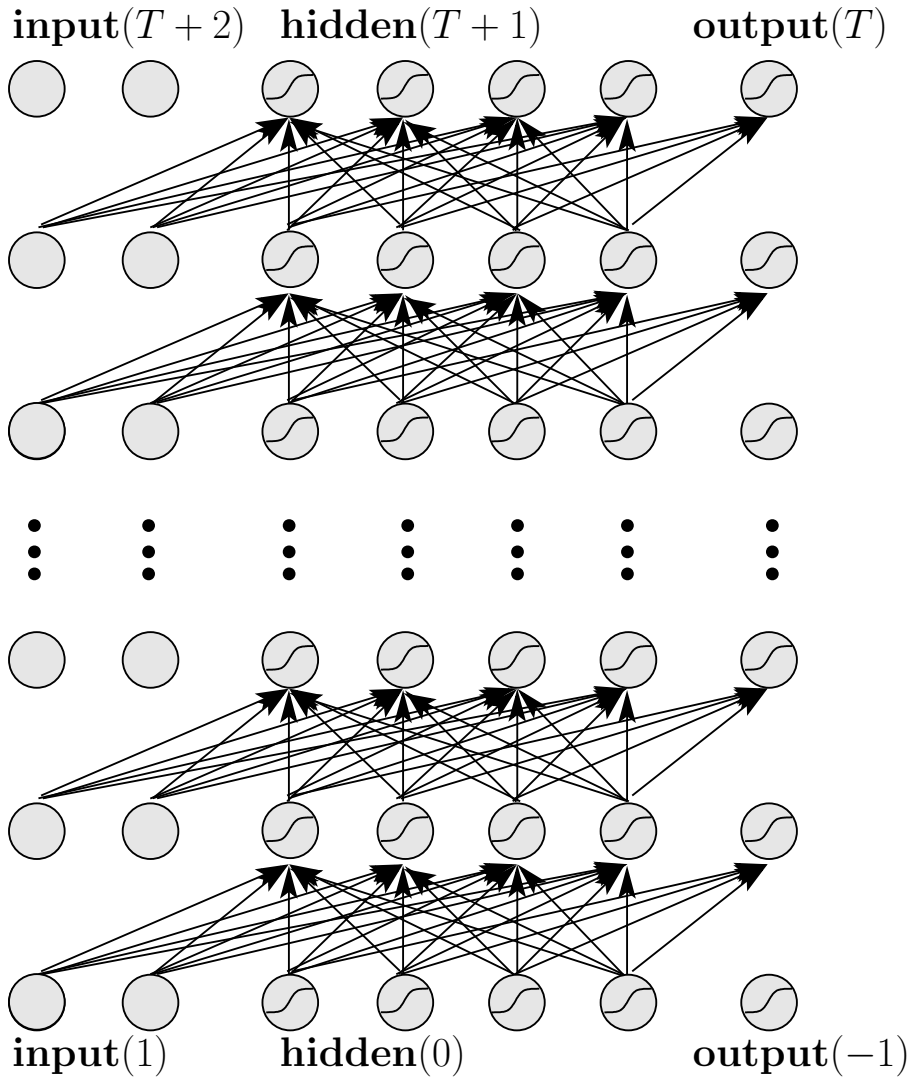


Figure 2.2: The recurrent network from Figure 2.1 left unfolded in time. Dummy inputs at time $(t+1)$ and $(t+2)$ and dummy outputs at $(t-2)$ and $(t-1)$ are used. The input is shifted two time steps against the output to allow the input information to be propagated through the network.

2.2.1 BPTT for the fully recurrent network

Let $\boldsymbol{w} \in \mathbb{R}^{I(D+2K)}$ be a vector that collects all entries of $\boldsymbol{W}, \boldsymbol{R}$, and $\boldsymbol{V}$ in an arbitrary but fixed order. To obtain update rules for the weights $\boldsymbol{w}$, we need to determine the gradient

$$\nabla_{\boldsymbol{w}} L = \frac{\partial}{\partial \boldsymbol{w}^\top} L\left(\boldsymbol{y}(1{:}T), g\left(\boldsymbol{a}(0), \boldsymbol{x}(1{:}T); \boldsymbol{w}\right)\right) = \frac{\partial L}{\partial \boldsymbol{w}^\top}. \tag{2.4}$$

Note that we dropped the index n, which enumerates the samples in our data set. That is, we derive the gradient for a general sample $(\boldsymbol{y}(1{:}T),\boldsymbol{x}(1{:}T))$. Once the gradient is known, we can apply the gradient descent update rule

$$\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} - \eta \nabla_{\boldsymbol{w}} L. \tag{2.5}$$

for some learning rate $\eta > 0$.

In the fully RNN, the gradient $\nabla_{\boldsymbol{w}} L$ decomposes into three parts corresponding to the weight matrices $\boldsymbol{W} = (w_{di}), \boldsymbol{R} = (r_{ij}), \boldsymbol{V} = (v_{ik})$. Let us begin with the output layer, that is v_{ik}, and consider an arbitrary time step t. The derivative is obtained by the chain rule and has the form

$$\frac{\partial L}{\partial v_{ik}} = \sum_{t=1}^{T} \frac{\partial L(\boldsymbol{y}(t), \hat{\boldsymbol{y}}(t))}{\partial v_{ik}} = \sum_{t=1}^{T} \frac{\partial L}{\partial \hat{y}_k(t)} \frac{\partial \hat{y}_k(t)}{\partial v_{ik}} \,. \tag{2.6}$$

The term $\partial L/(\partial \hat{y}_k(t))$ is the derivative of the loss with respect to the k-th component of the model output. We will call this term $e_k(t)$. To find this derivative we only need to know the loss function and the model output but not the model itself. For instance, if we choose the halved squared error as loss function

$$L(\boldsymbol{y}(t), \hat{\boldsymbol{y}}(t)) = \frac{1}{2} \left\| \boldsymbol{y}(t) - \hat{\boldsymbol{y}}(t) \right\|_2^2 \qquad \text{then} \qquad e_k(t) = \frac{\partial L}{\partial \hat{y}_k(t)} = \hat{y}_k(t) - y_k(t) \,. \tag{2.7}$$

The term $\partial \hat{y}_k(t)/(\partial v_{ik})$ is the derivative of the k-th model output with respect to the weight v_{ik}, which has the form

$$\hat{y}_k(t) = \varphi\left(\sum_{j=1}^{I} v_{jk} a_j(t)\right) \tag{2.8}$$

and consequently

$$\frac{\partial \hat{y}_k(t)}{\partial v_{ik}} = \varphi'\left(\sum_{j=1}^{I} v_{jk} a_j(t)\right) a_i(t). \tag{2.9}$$

If we put this together we arrive at the gradient for v_{ik}

$$\frac{\partial L}{\partial v_{ik}} = \sum_{t=1}^{T} e_k(t)\, \varphi'\left(\sum_{j=1}^{I} v_{jk} a_j(t)\right) a_i(t) \tag{2.10}$$

and the gradient descent update rule for v_{ik} is

$$v_{ik}^{\text{new}} = v_{ik}^{\text{old}} - \eta \frac{\partial L}{\partial v_{ik}}, \tag{2.11}$$

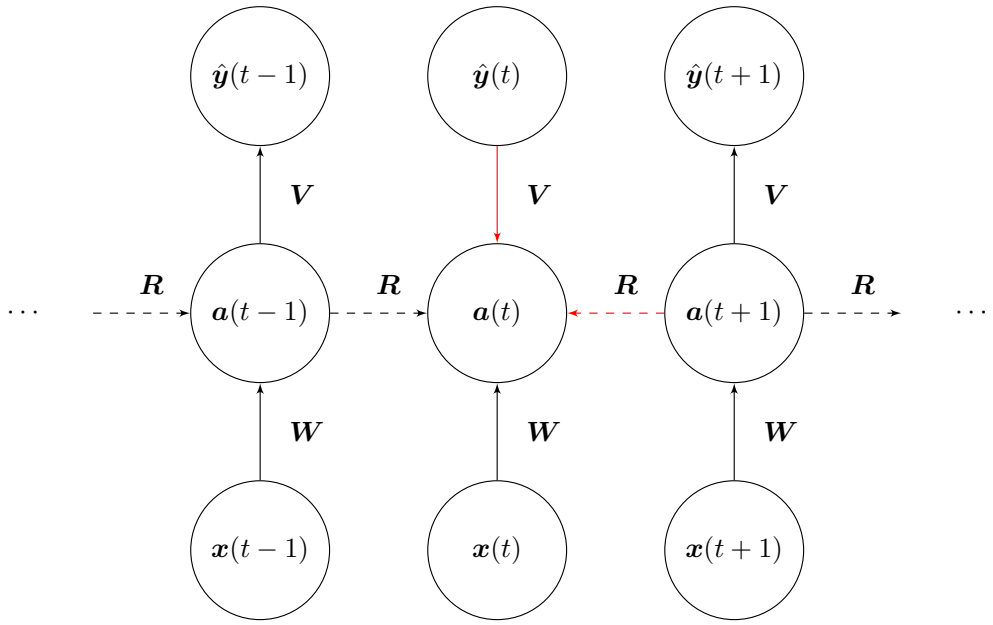


Figure 2.3: The recurrent network from Figure 1.4 unfolded in time. The red arrows indicate the contributions to $\boldsymbol{\delta}(t)$ from the presence (i.e. from $\hat{\boldsymbol{y}}(t)$ to $\boldsymbol{a}(t)$) and from the future (i.e. from $\boldsymbol{a}(t+1)$ to $\boldsymbol{a}(t)$). This is used in Equation (2.12).

where η is the learning rate. Equation (2.11) reveals a disadvantage of BPTT: the activations $\boldsymbol{a}(t)$ must be stored over the whole sequence. This can cause heavy memory consumption for long sequences (i.e. large T).

Before we look at the derivatives with respect to r_{ij} and w_{id} we introduce an ancillary concept: Derivatives of the loss with respect to pre-activations are called *deltas*. Deltas are helpful in backpropagation because they enable a recursive calculation of the gradients through time.

In our RNN example, described by Equation (1.9), there is only one hidden pre-activation vector at each time step, namely $\boldsymbol{s}(t)$. We define

$$\begin{aligned}
\boldsymbol{\delta}(t)^\top &= \frac{\partial L}{\partial \boldsymbol{s}(t)} = \frac{\partial L}{\partial \boldsymbol{a}(t)} \frac{\partial \boldsymbol{a}(t)}{\partial \boldsymbol{s}(t)} \\
&= \left(\frac{\partial L(\boldsymbol{y}(t), \hat{\boldsymbol{y}}(t))}{\partial \boldsymbol{a}(t)} + \frac{\partial L}{\partial \boldsymbol{s}(t+1)} \frac{\partial \boldsymbol{s}(t+1)}{\partial \boldsymbol{a}(t)} \right) \frac{\partial \boldsymbol{a}(t)}{\partial \boldsymbol{s}(t)} \\
&= \left(\frac{\partial L}{\partial \hat{\boldsymbol{y}}(t)} \frac{\partial \hat{\boldsymbol{y}}(t)}{\partial \boldsymbol{a}(t)} + \frac{\partial L}{\partial \boldsymbol{s}(t+1)} \frac{\partial \boldsymbol{s}(t+1)}{\partial \boldsymbol{a}(t)} \right) \frac{\partial \boldsymbol{a}(t)}{\partial \boldsymbol{s}(t)} \\
&= \left(\boldsymbol{e}(t)^\top \operatorname{diag}(\varphi'(\boldsymbol{V}^\top \boldsymbol{a}(t))) \boldsymbol{V}^\top + \boldsymbol{\delta}(t+1)^\top \boldsymbol{R}^\top \right) \operatorname{diag}(f'(\boldsymbol{s}(t))).
\end{aligned} \tag{2.12}$$

Note that $\boldsymbol{\delta}(t)$ depends on $\boldsymbol{\delta}(t+1)$. This is a recursion into the future, i.e. we have to initialize the recursion at time T and then work our way backwards through time. Figure 2.3 depicts the dependencies of $\boldsymbol{\delta}(t)$ on the presence and the future.

To keep track of weight sharing, i.e. to account for the fact that $\boldsymbol{W}$ and $\boldsymbol{R}$ are reused at every time step, we equip the weights with a time index to keep track of their usages during the forward

pass, i.e.

$$\boldsymbol{s}(t) = \boldsymbol{W}(t)^\top \boldsymbol{x}(t) + \boldsymbol{R}(t)^\top \boldsymbol{a}(t-1). \tag{2.13}$$

With the help of the deltas, we can also determine the gradients for the recurrent weights as

$$\begin{aligned} \frac{\partial L}{\partial r_{ij}} &= \sum_{t=1}^{T} \frac{\partial L}{\partial r_{ij}(t)} = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{s}(t)} \frac{\partial \boldsymbol{s}(t)}{\partial r_{ij}(t)} \\ &= \sum_{t=1}^{T} \frac{\partial L}{\partial s_i(t)} \frac{\partial s_i(t)}{\partial r_{ij}(t)} = \sum_{t=1}^{T} \delta_i(t) a_j(t-1). \end{aligned} \tag{2.14}$$

Similarly, we can write the input weight gradients as

$$\begin{aligned} \frac{\partial L}{\partial w_{id}} &= \sum_{t=1}^{T} \frac{\partial L}{\partial w_{id}(t)} = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{s}(t)} \frac{\partial \boldsymbol{s}(t)}{\partial w_{id}(t)} \\ &= \sum_{t=1}^{T} \frac{\partial L}{\partial s_i(t)} \frac{\partial s_i(t)}{\partial w_{id}(t)} = \sum_{t=1}^{T} \delta_i(t) x_d(t) \end{aligned} \tag{2.15}$$

This completes the set of tools needed for the backward pass. The gradient descent update rules are

$$r_{ij}^{\text{new}} = r_{ij}^{\text{old}} - \eta \frac{\partial L}{\partial r_{ij}} \qquad \text{and} \qquad w_{di}^{\text{new}} = w_{di}^{\text{old}} - \eta \frac{\partial L}{\partial w_{di}} \tag{2.16}$$

respectively.

Outer product representation Although the equations for the backward pass are already complete, we will now give an alternative form, that allows us to express the gradients with respect to the whole weight matrices in a compact manner. That is, instead of computing the gradient of the loss with respect to one entry v_{ik} we will now write the gradient with respect to the entire matrix $\boldsymbol{V}$ using outer products of vectors. First we give another auxiliary definition by

$$\boldsymbol{\psi}(t)^\top = \frac{\partial L}{\partial \boldsymbol{V}^\top \boldsymbol{a}(t)} = \boldsymbol{e}(t)^\top \odot \varphi'\left(\boldsymbol{a}(t)^\top \boldsymbol{V}\right), \tag{2.17}$$

where $\odot$ denotes Hadamard's vector product, which defined as $(\boldsymbol{x} \odot \boldsymbol{y})_i = x_i\, y_i$.

Reconsidering Equation (2.10) and using the definition (2.17) we find that the gradient $\partial L(\hat{\boldsymbol{y}}(t), \boldsymbol{y}(t))/(\partial v_{ik})$ can be written as a product of two numbers $\psi_k(t)$ and $a_i(t)$. To compute this for the whole matrix $\boldsymbol{V} \in \mathbb{R}^{I \times K}$ we have to construct this product once for all possible combinations of $i \in \{1, \ldots, I\}$ and $k \in \{1, \ldots, K\}$. We can represent this in vector notation by an outer product and write

$$\frac{\partial L}{\partial \boldsymbol{V}} = \sum_{t=1}^{T} \boldsymbol{\psi}(t)\, \boldsymbol{a}(t)^\top, \tag{2.18}$$

We can apply the same trick to the gradients with respect to $\boldsymbol{W}$ and $\boldsymbol{R}$, which yields

$$\frac{\partial L}{\partial \boldsymbol{R}} = \sum_{t=1}^{T} \boldsymbol{\delta}(t)\, \boldsymbol{a}(t-1)^\top \tag{2.19}$$

$$\frac{\partial L}{\partial \boldsymbol{W}} = \sum_{t=1}^{T} \boldsymbol{\delta}(t)\, \boldsymbol{x}(t)^\top. \tag{2.20}$$

Batched learning and matrix product representation When we derived the backpropagation update rules we started by considering only a single sample $(\boldsymbol{x}, \boldsymbol{y})$. In practice, however, we are given a data set

$$\left\{\boldsymbol{x}(1{:}T)^{(n)}, \boldsymbol{y}(1{:}T)^{(n)}\right\}_{n=1}^{N} \tag{2.21}$$

of N samples. Now we can randomly choose a sample from the data set and feed it into our network. After computing the loss and the resulting gradients, we can apply the update rules (2.11) and (2.16) to the network parameters. After this, we choose another sample and feed it into our network, and so on. This procedure is known as *online learning*.

Sometimes it is appealing to make update steps depending on more than one sample. Often, this is computationally more efficient with respect to the whole data set as both forward and backward pass can be computed in parallel for multiple samples at once. Using the whole data set for computing an update step is known as *full batch learning* or simply *batch learning*. For most practically relevant data sets and network architectures this is infeasible. Moreover, the stochasticity in the gradient that comes from selecting samples at random is important for learning because it helps escaping from plateaus in the loss landscape.

The compromise between online learning and batch learning is called *mini-batch learning*. It is the most commonly applied learning technique among those three, where one chooses a small set of samples from the data set randomly. This set is called a *mini-batch* and is used to compute the gradients for the next update step. Mini-batch learning combines the advantages of online learning (stochasticity) and batch learning (parallelizability).

In the following, we will extend the mathematical framework to batches of samples. We consider batch learning rather than mini-batch learning but only for notational convenience. Instead of an input vector $\boldsymbol{x}(t)$ we now have an input matrix $\boldsymbol{X}(t) \in \mathbb{R}^{D \times N}$, that combines N input vectors

$$\begin{aligned}\boldsymbol{X}(t) &= \begin{pmatrix}\boldsymbol{x}(t)^{(1)} & \boldsymbol{x}(t)^{(2)} & \cdots & \boldsymbol{x}(t)^{(N)}\end{pmatrix} \\ &= \begin{pmatrix} x_1(t)^{(1)} & x_1(t)^{(2)} & \cdots & x_1(t)^{(N)} \\ x_2(t)^{(1)} & x_2(t)^{(2)} & \cdots & x_2(t)^{(N)} \\ \vdots & \vdots & \ddots & \vdots \\ x_D(t)^{(1)} & x_D(t)^{(2)} & \cdots & x_D(t)^{(N)} \end{pmatrix}.\end{aligned} \tag{2.22}$$

The forward pass for the fully recurrent network becomes

$$\begin{aligned}\boldsymbol{S}(t) &= \boldsymbol{W}^\top \boldsymbol{X}(t) + \boldsymbol{R}^\top \boldsymbol{A}(t-1) \\ \boldsymbol{A}(t) &= f(\boldsymbol{S}(t)) \\ \hat{\boldsymbol{Y}}(t) &= \varphi(\boldsymbol{V}^\top \boldsymbol{A}(t)),\end{aligned} \tag{2.23}$$

where the (pre-)activation vectors $\boldsymbol{s}(t), \boldsymbol{a}(t), \boldsymbol{y}(t)$ have become matrices $\boldsymbol{S}(t), \boldsymbol{A}(t), \boldsymbol{Y}(t)$, respectively, in the same manner as $\boldsymbol{x}(t)$ became $\boldsymbol{X}(t)$. Further, construct $\boldsymbol{\Psi}(t)$ from $\boldsymbol{\psi}(t)$ as well as $\boldsymbol{\Delta}(t)$ from $\boldsymbol{\delta}(t)$ similarly. Note that the information carried by distinct samples remains strictly separate, i.e. samples do not interfere but are processed in parallel. This means if the content of the n-th input $\boldsymbol{x}(1{:}T)^{(n)}$ is changed, then this affects only the n-th output $\hat{\boldsymbol{y}}(1{:}T)^{(n)}$ while all other columns in the output matrices $\hat{\boldsymbol{Y}}(1), \ldots \hat{\boldsymbol{Y}}(T)$, remain unchanged.

Equation (2.1) tells us to combine the loss values of N samples into an empirical risk value by taking their mean. Therefore, we also have to take means when computing the gradients.

Reconsider the outer product representations (2.18) and (2.20). The gradients of the empirical risk with respect to the weights can be written as

$$\frac{\partial R_{\text{emp}}}{\partial \boldsymbol{V}} = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T} \boldsymbol{\psi}(t)^{(n)} \left(\boldsymbol{a}(t)^{(n)} \right)^{\top} = \frac{1}{N} \sum_{t=1}^{T} \boldsymbol{\Psi}(t) \, \boldsymbol{A}(t)^{\top} \tag{2.24}$$

$$\frac{\partial R_{\text{emp}}}{\partial \boldsymbol{R}} = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T} \boldsymbol{\delta}(t)^{(n)} \left(\boldsymbol{a}(t-1)^{(n)} \right)^{\top} = \frac{1}{N} \sum_{t=1}^{T} \boldsymbol{\Delta}(t) \, \boldsymbol{A}(t-1)^{\top} \tag{2.25}$$

$$\frac{\partial R_{\text{emp}}}{\partial \boldsymbol{W}} = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T} \boldsymbol{\delta}(t)^{(n)} \left(\boldsymbol{x}(t)^{(n)} \right)^{\top} = \frac{1}{N} \sum_{t=1}^{T} \boldsymbol{\Delta}(t) \, \boldsymbol{X}(t)^{\top} \tag{2.26}$$

because the sum over the sample index n can be reformulated as a matrix product. For mini-batch learning, the only adaptation needed is to let N denote the mini-batch size instead of the data set size such that n, consequently, indexes the samples within the mini batch instead of the whole data set.

2.3 Regularization

LSTM networks were the first recurrent architectures to successfully solve complicated problems with long-term dependencies. This, however, highlighted a new problem that was already well-known from other machine learning disciplines: *overfitting*. In other words, now that the networks were finally learning, the new challenge was to stall learning while the models are still generalizing well, i.e. before they start to overfit heavily.

2.3.1 Early stopping

If the amount of data available is large enough, then one of the easiest and also most effective regularization techniques is to split the data into three parts, a training set, a validation set, and a test set. During training on the training set, one constantly evaluates the model perfomance on the validation set. As soon as the validation error becomes significantly worse compared to the training error, we simply stop training. Finally, we use the test set to estimate the generalization error. This simple regularization method is called *early stopping*.

2.3.2 Weight decay

One way of formalizing the problem of overfitting is via model complexity. The model selection procedure should select the simplest model that is still complex enough to solve the task (Vapnik, 1998). In neural networks, the norm $\|\boldsymbol{w}\|$ of the weights is a common measure of complexity.

Following this thought, a very natural way to obtain a regularization technique is to impose a zero-mean Gaussian prior on the weights, i.e.

$$\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{0}, \sigma^2 \boldsymbol{I}). \tag{2.27}$$

In the following, let $\boldsymbol{z} = (\boldsymbol{x}, \boldsymbol{y})$ denote a data sample consisting of input $\boldsymbol{x}$ and label $\boldsymbol{y}$. We obtain the log-posterior

$$\begin{aligned}\log p(\boldsymbol{w} \mid \boldsymbol{z}) &\propto \log p(\boldsymbol{z} \mid \boldsymbol{w}) + \log p(\boldsymbol{w}) \\ &= \log p(\boldsymbol{z} \mid \boldsymbol{w}) + \log \left((2\pi)^{-\frac{\dim(\boldsymbol{w})}{2}} \det(\sigma^2 \boldsymbol{I})^{-\frac{1}{2}} e^{-\frac{1}{2}\boldsymbol{w}^\top (\sigma^2 \boldsymbol{I})^{-1} \boldsymbol{w}} \right) \\ &= \log p(\boldsymbol{z} \mid \boldsymbol{w}) - \frac{1}{2} \left(\dim(\boldsymbol{w}) \log(2\pi\sigma^2) + \frac{1}{\sigma^2} \boldsymbol{w}^\top \boldsymbol{w} \right).\end{aligned} \tag{2.28}$$

Since the term $(1/2)\dim(\boldsymbol{w})\log(2\pi\sigma^2)$ does not depend on the values of $\boldsymbol{w}$ but only on its dimension, it is just a constant offset to the optimization objective and it is equivalent to maximizing

$$\log p(\boldsymbol{w} \mid \boldsymbol{z}) \propto \log p(\boldsymbol{z} \mid \boldsymbol{w}) - \frac{1}{2\sigma^2} \boldsymbol{w}^\top \boldsymbol{w} = \log p(\boldsymbol{z} \mid \boldsymbol{w}) - \lambda \sum_{i=1}^{\dim(\boldsymbol{w})} w_i^2. \tag{2.29}$$

Note that $\log(\boldsymbol{z} \mid \boldsymbol{w})$ is just the likelihood function of the data $\boldsymbol{z}$ given parameters $\boldsymbol{w}$, i.e. this is our main objective. Imposing a Gaussian prior on the weights $\boldsymbol{w}$ adds a *regularization term* to our main objective that aims to keep the squares of the weights small. By choosing an appropriate variance σ^2 for the prior distribution, we can weigh the regularization term against our main objective $\log(\boldsymbol{z} \mid \boldsymbol{w})$. Often, we write $\lambda = 1/(2\sigma^2)$ for brevity.

Similarly, one can impose a Laplacian prior on the weights, i.e. $\boldsymbol{w} \sim \mathrm{Laplace}(0, \mathrm{bI})$. This leads to the regularization term

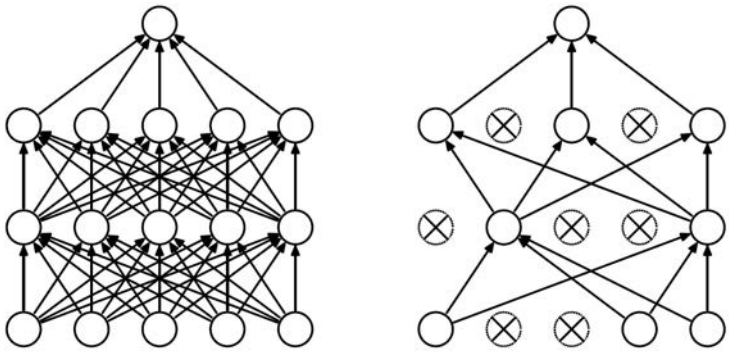
$$\lambda \sum_{i=1}^{\dim(\boldsymbol{w})} |w_i| \,. \tag{2.30}$$

These techniques are not exclusively used for recurrent neural networks. They are not even limited to neural networks but belong to the most widely used regularization techniques in machine learning. They are applicable whenever $\|\boldsymbol{w}\|$ can be interpreted as a term measuring model complexity.

2.3.3 Dropout

Dropout (Srivastava et al., 2014) is a very successful regularization technique for feedforward networks. The basic idea is to prevent co-adaption of units in the network by randomly switching some of the units off, i.e. setting their activation values to zero, while training. Therefore, a unit cannot learn to rely on other units to function properly and the units are encouraged to operate independently, which implies a certain robustness of the prediction of the network. Figure 2.4 shows how dropout works on fully connected feedforward nets.

Now we could try to apply the same procedure to recurrent neural networds. If we unfold the RNN in time we can apply dropout in the same manner as we would for feedforward networks. This includes dropping out the recurrent activations $\boldsymbol{a}(t)$ for $t \in 1, \ldots, T$. Recall that $\boldsymbol{a}(t)$ can be viewed as the network's memory over time. This means, when dropping out $a_i(t)$ at time t, then all information gathered up until time t is lost for the remainder of the sequence. If this is done on every time step, then most recurrent units will have been dropped out by the end of the sequence, at least for non-trivial sequence lengths T. We will later see some techniques to avoid this problem.



(a) Standard Neural Net (b) After applying dropout.

Figure 2.4: Left: A feedforward net without dropout. All neurons are active. Right: The same network with dropout. A fraction of the units have been dropped out, which is indicated by a cross. Source: Srivastava et al. (2014)

2.4 Other RNN learning algorithms

As we have seen, BPTT requires to compute the gradients with respect to all past time steps. That is, for every network output $\hat{\boldsymbol{y}}(t)$, we have to determine the gradient with respect to all previous time steps $1, \ldots, t-1$ additionally to that at t. This requires storage of all activations at all time steps and results in a both memory and computationally intensive learning algorithm. The complexity of BPTT is $\mathcal{O}(T \dim \boldsymbol{w})$.

2.4.1 Truncated backpropagation through time

Due to the high complexity of BPTT, other learning strategies have been considered. Truncated backpropagation through time (Williams and Peng, 1990) introduces a maximum number of time steps for both forward and backward pass during learning. That is, input sequences are divided into chunks of length τ. The RNN processes a chunk and an update step is performed. This is denoted BPTT(τ). This introduces a maximal range of dependencies which can be directly captured by the gradient. Therefore, the choice of τ is critical and depends strongly on the learning task. The complexity of truncated BPTT is $\mathcal{O}(\tau \dim \boldsymbol{w})$.

Truncated BPTT only approximates the gradient since functional dependencies from the forward pass are cut. Therefore, training via truncated BPTT is not guaranteed to converge. The truncated gradient and the true gradient may point in different directions (Tallec and Ollivier, 2018b).

2.4.2 Real-time recurrent learning

Real-time recurrent learning (RTRL; Werbos, 1981; Robinson and Fallside, 1987; Williams and Zipser, 1989; Gherrity, 1989) computes all contributions to the gradients during the forward pass already. Thus, activations need not be stored along the whole sequence. Consequently, the complexity of RTRL is independent of the sequence length T. Therefore, RTRL is an alternative when

dealing with very long sequences. The key idea is to pass the gradients from the last time step recursively into the present. However, this comes at the cost of increased memory and compute consumption depending on the number of parameters in the network.

Reconsider the forward dynamics of the pre-activations with time-dependent weights (cf. Equation (2.13)), which we restate here for convenience.

$$\boldsymbol{s}(t) = \boldsymbol{W}(t)^\top \boldsymbol{x}(t) + \boldsymbol{R}(t)^\top \boldsymbol{a}(t-1) \tag{2.31}$$

We now drop the time index t from the network parameters and consider only the n-th neuron, i.e.

$$s_n(t) = \sum_{l=1}^{D} w_{ln} x_l(t) + \sum_{k=1}^{I} r_{kn} a_k(t-1). \tag{2.32}$$

Dropping the time index means that we consider the weights as shared in time and consequently we have to incorporate the fact that $\boldsymbol{a}(t-1)$ and, therefore, $\boldsymbol{s}(t-1)$ also depend on $\boldsymbol{W}$ and $\boldsymbol{R}$. Now we can give a recursive form of the derivative w.r.t. the network parameters by

$$\begin{aligned} \frac{\partial s_n(t)}{\partial r_{ij}} &= \sum_{k=1}^{I} \frac{\partial r_{kn}}{\partial r_{ij}} a_k(t-1) + \sum_{k=1}^{I} r_{kn} \frac{\partial a_k(t-1)}{\partial r_{ij}} \\ &= a_i(t-1)[n=j] + \sum_{k=1}^{I} r_{kn} f'\big(s_k(t-1)\big) \frac{\partial s_k(t-1)}{\partial r_{ij}} \end{aligned} \tag{2.33}$$

and

$$\begin{aligned} \frac{\partial s_n(t)}{\partial w_{dj}} &= \sum_{l=1}^{D} \frac{\partial w_{ln}}{\partial w_{dj}} x_l(t) + \sum_{k=1}^{I} r_{kn} \frac{\partial a_k(t-1)}{\partial w_{dj}} \\ &= x_d(t)[n=j] + \sum_{k=1}^{I} r_{kn} f'\big(s_k(t-1)\big) \frac{\partial s_k(t-1)}{\partial w_{dj}}, \end{aligned} \tag{2.34}$$

where $[n = j]$ is the Iverson bracket that evaluates logical expressions[1], i.e. $[A] = 1$ if A is true and $[A] = 0$ otherwise. The recursive structure of Equations (2.33, 2.34) enables us to compute the gradients during the forward pass already. We initialize the recursions with

$$s_k(0) = 0 \qquad \text{and consequently} \qquad \frac{\partial s_k(0)}{\partial w_{dj}} = \frac{\partial s_k(0)}{\partial r_{ij}} = 0 \qquad \text{for all } d, i, j, k. \tag{2.35}$$

Note that the partial derivatives $\partial \boldsymbol{s}(t)/\partial \boldsymbol{R}, \partial \boldsymbol{s}(t)/\partial \boldsymbol{W}$ can stand alone, i.e. they can be computed as soon as $\boldsymbol{s}(t)$ is activated and before the network output $\hat{\boldsymbol{y}}(t)$ is activated or the loss $L(\boldsymbol{y}(t), \hat{\boldsymbol{y}}(t))$ is evaluated. To perform a weight update, of course, $\partial \boldsymbol{s}(t)/\partial \boldsymbol{R}, \partial \boldsymbol{s}(t)/\partial \boldsymbol{W}$ must be compiled into the context of a loss derivative.

At every time step t, RTRL must store the derivatives for computing the gradients for the next time step $t+1$. This makes the complexity of RTRL independent of the sequence length T while using correct (as opposed to truncated BPTT) gradients at each time step. The disadvantage is that $\partial \boldsymbol{s}(t)/\partial \boldsymbol{R}$ is a tensor of shape $I \times I \times I$, which is cubic in the number of hidden neurons. When updating one entry in this cubic tensor we have to evaluate and sum up I items at every time step and therefore RTRL has a computational complexity of $\mathcal{O}(T\, I^4)$. This is intractable for most RNNs of reasonable size. Some approaches have been put forward to alleviate this issue.

[1] We could have used the Kronecker delta here, but that introduces notation conflicts with error deltas.

2.4.3 Schmidhuber's Approach

Jürgen Schmidhuber suggested an algorithm with complexity $\mathcal{O}(I^3)$, which is exact and not truncated and is between $\mathcal{O}(TI^2)$ of BPTT and $\mathcal{O}(I^4)$ of RTRL (Schmidhuber, 1991, 1992).

The idea is to divide the sequence in blocks of length I. Within each block BPTT is performed with an update complexity of $\mathcal{O}(I^3)$. Between the blocks RTRL is performed to collect all gradient information from the blocks with $\mathcal{O}(I^4)$ complexity. However RTRL is performed only every I time steps, the complexity per time step is only $\mathcal{O}(I^3)$. That is for every BPTT block with subsequent RTRL we have complexity $\mathcal{O}(I^3)$. Since we have T/I such blocks, the final complexity is that of BPTT. However, only the activations of sequences of length I have to be stored.

2.5 The vanishing gradient problem

2.5.1 Long-term dependencies

Even if BPTT is applied correctly to Simply Recurrent Networks the learning signal degrades after few time steps of backpropagation. In the following, we will analyze why this happens and this will lead us towards a solution of this problem.

To formalize the concept of a long-term dependency, let us consider two points in time t' and t where $t' \ll t$. Further, assume that the input $\boldsymbol{x}(t')$ at time t' carries essential information for inferring the correct output $\boldsymbol{y}(t)$ at time t. Then the network must learn to store this information at time t', keep it in memory for $t - t'$ time steps, and finally wire this piece of information to the output at time t. This is a non-trivial learning problem which turned out to be too hard to solve for architectures like the fully recurrent network and large enough time spans $t - t'$.

A fundamental precondition for learning such long-term dependencies is that the gradient

$$\frac{\partial L(\boldsymbol{y}(t), \hat{\boldsymbol{y}}(t))}{\partial \boldsymbol{W}(t')} = \frac{\partial L(\boldsymbol{y}(t), \hat{\boldsymbol{y}}(t))}{\partial \boldsymbol{s}(t)} \frac{\partial \boldsymbol{s}(t)}{\partial \boldsymbol{s}(t')} \frac{\partial \boldsymbol{s}(t')}{\partial \boldsymbol{W}(t')} \tag{2.36}$$

must be large enough for the model to learn a dependence of the output $\hat{\boldsymbol{y}}(t)$ on the input $\boldsymbol{x}(t')$ in the first place and develop a strategy to store its information content. The crucial part is the middle factor on the right-hand side of Equation (2.36) because it carries the gradient backwards over $t - t'$ time steps. Let us first consider the somewhat simpler problem of going only one step backwards in time, i.e. $t' = t - 1$, and define

$$\iota_{ij}(t) = \frac{\partial s_i(t)}{\partial s_j(t-1)} = r_{ij}(t) f'(s_j(t-1)) \,, \tag{2.37}$$

that is the derivative of a pre-activation $s_i(t)$ with respect to its previous pre-activation $s_j(t-1)$. Further, we define the matrix

$$\boldsymbol{J}(t) = \frac{\partial \boldsymbol{s}(t)}{\partial \boldsymbol{s}(t-1)} = (\iota_{ij}(t)) \in \mathbb{R}^{I \times I}. \tag{2.38}$$

If we want to go l time steps backwards, i.e. we have $t' = t - l$, the chain rule decomposes the problem as follows:

$$\frac{\partial s_i(t)}{\partial s_j(t-l)} = \frac{\partial s_i(t)}{\partial \boldsymbol{s}(t-1)} \frac{\partial \boldsymbol{s}(t-1)}{\partial \boldsymbol{s}(t-2)} \cdots \frac{\partial \boldsymbol{s}(t-l+1)}{\partial s_j(t-l)} = \left(\prod_{\tau=0}^{l} \boldsymbol{J}(t-\tau) \right)_{ij} \tag{2.39}$$

2.5.2 Vanishing and exploding gradients

Although BPTT uses the correct gradients for weight updates it suffers from a fundamental problem when applied to the fully recurrent network. The gradients practically fail to carry the error signal over more than a few steps backwards in time. This shortcoming is known as the *vanishing gradient problem* and was first formalized by Hochreiter (1991) and later also by Bengio et al. (1994). For instance, consider the fully recurrent architecture with only one hidden neuron. That is we can write the forward pass as

$$\begin{aligned} s(t) &= \boldsymbol{u}^\top \boldsymbol{x}(t) + ra(t-1) \\ a(t) &= f(s(t)) \\ \hat{\boldsymbol{y}}(t) &= \boldsymbol{v}a(t) \end{aligned} \tag{2.40}$$

and we have only one recurrent weight r. We write $\boldsymbol{u}$ to denote the input weights in vector form because $\boldsymbol{w}$ already denotes the whole parameter set. Reconsider Equation (2.37), where the gradient is propagated one time step backwards. For architecture (2.40) it simplifies to

$$\iota(t) = \frac{\partial s(t)}{\partial s(t-1)} = r(t) f'(s(t-1)) = r f'(s(t-1)) \tag{2.41}$$

because $r(1) = r(2) = \cdots = r(T) = r$. Using this to adapt Equation (2.39) leads to

$$\begin{aligned} \frac{\partial s(t)}{\partial s(t-l)} &= \frac{\partial s(t)}{\partial s(t-1)} \frac{\partial s(t-1)}{\partial s(t-2)} \cdots \frac{\partial s(t-l+1)}{\partial s(t-l)} \\ &= \prod_{t'=0}^{l-1} \iota(t-t') = r^l \prod_{t'=1}^{l} f'(s(t-t')) \,. \end{aligned} \tag{2.42}$$

It is easy to see that (2.42) is generally unstable for large l depending on the choice of r and f. In case $r\, f'(s(t-t')) < 1$ the gradient decays exponentially over time, i.e. the gradient *vanishes*, and if $r\, f'(s(t-t')) > 1$ the gradient grows exponentially over time, i.e. the gradient *explodes*. Both cases are catastrophic for learning. With vanishing gradients the weight updates become infinitesimally small, in other words the weights do not change and learning stalls. With exploding gradients the absolute values of the weights quickly become larger than what a computer can represent and we run into overflows. Only the special case $r\, f'(s(t-t')) = 1$ is numerically stable over long time spans.

Solving the differential equation $f'(x) = 1/r$ leads to the activation function $f(x) = x/r$, which is linear and its slope compensates for the decay introduced by the factor r. Therefore, without loss of generality we can choose $r = 1$ which leads to a constant recurrent weight and the identity activation $f(x) = x$. Figure 2.5 shows a recurrent unit that fulfills $f(x) = x$ and $r = 1$ and thereby avoids the vanishing gradient problem. These conditions, however, lead to a linear model with constant recurrent parameters, i.e. such networks have very limited learning capabilities. Approaches to resolve this conflict will be presented later.

Adding more hidden units does not alleviate the vanishing gradient problem but only makes it somewhat harder to see. Equation (2.39) generally suffers from the same instabilities. In particular, Bengio et al. (1994) showed that under some conditions which are considered necessary for storing information over longer periods of time all eigenvalues of $\boldsymbol{J}(t) = \partial \boldsymbol{s}(t)/(\partial \boldsymbol{s}(t-l))$ must lie inside

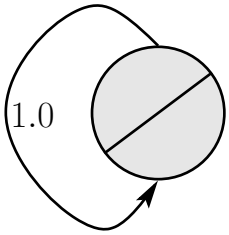


Figure 2.5: A single unit with self-recurrent connection which avoids the vanishing gradient.

the unit circle, which in turn implies that $\lim_{l\to\infty} \partial \boldsymbol{s}(t)/(\partial \boldsymbol{s}(t-l)) = 0, t \geqslant l$ with exponentially fast convergence.

In the following we will determine the properties we have to require for $\boldsymbol{R}$ to guarantee stable gradients in the fully recurrent network (1.9). As stated above, the only activation function with stable gradient over an arbitrary number of time steps is the identity function $f(x) = x$ (or an affine version of it). This choice simplifies Equation (2.37) to

$$\iota_{ij}(t) = \frac{\partial s_i(t)}{\partial s_j(t-1)} = r_{ij}(t) \tag{2.43}$$

because $f'(x) = 1$ for all $x \in \mathbb{R}$. As a consequence, Equation (2.39) changes to

$$\frac{\partial s_i(t)}{\partial s_j(t-l)} = \left(\boldsymbol{R}^l\right)_{ij}. \tag{2.44}$$

Now define

$$\boldsymbol{d}(t)^\top = \frac{\partial L(\boldsymbol{y}(t), \hat{\boldsymbol{y}}(t))}{\partial \boldsymbol{s}(t)} \tag{2.45}$$

and consider the partial derivative

$$\frac{\partial L(\boldsymbol{y}(t), \hat{\boldsymbol{y}}(t))}{\partial \boldsymbol{s}(t-l)} = \boldsymbol{d}(t)^\top \frac{\partial \boldsymbol{s}(t)}{\partial \boldsymbol{s}(t-l)} = \boldsymbol{d}(t)^\top \boldsymbol{R}^l. \tag{2.46}$$

The vector $\boldsymbol{d}(t)$ is the gradient that comes from the output layer at time step t. One way to obtain a stable system is to preserve the magnitude of $\boldsymbol{d}(t)$ in the sense that $\|\boldsymbol{d}(t)\| = \left\|\boldsymbol{d}(t)^\top \boldsymbol{R}\right\|$. If this property does not hold the magnitude of $\left\|\boldsymbol{d}(t)^\top \boldsymbol{R}^l\right\|$ can either grow or shrink with l in exponential manner. In other words, we could have exploding or vanishing gradients, respectively.

So the desired property is $\|\boldsymbol{d}(t)\| = \left\|\boldsymbol{d}(t)^\top \boldsymbol{R}\right\|$, i.e. $\boldsymbol{R}$ must be norm preserving because then also $\|\boldsymbol{d}(t)\| = \left\|\boldsymbol{d}(t)^\top \boldsymbol{R}^l\right\|$ holds inductively and the magnitude of the gradient is stable regardless of l. In Euclidean geometry, preserving the norm is equivalent to preserving the dot product, i.e. $\boldsymbol{R}$ must satisfy

$$(\boldsymbol{R}\boldsymbol{x})^\top (\boldsymbol{R}\boldsymbol{y}) = \boldsymbol{x}^\top \boldsymbol{R}^\top \boldsymbol{R}\boldsymbol{y} = \boldsymbol{x}^\top \boldsymbol{y} \qquad \text{for all } \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^I, \tag{2.47}$$

which is true if and only if $\boldsymbol{R}^\top \boldsymbol{R} = \boldsymbol{I}_I$, that is $\boldsymbol{R}$ must be an *orthogonal matrix*. Trivially, the identity $\boldsymbol{I}_I$ itself is one possible choice, which is made e.g. for LSTM (see Section 3). Another example that makes use of this insight is the work of Le et al. (2015) who suggest to initialize a ReLU-activated fully recurrent network with $\boldsymbol{I}_I$ as recurrent weight matrix to learn long-term dependencies.

The choice $\boldsymbol{R} = \boldsymbol{I}_I$ has a special property, which is of particular interest for recurrent neural networks. It allows for temporal generalization. If $\boldsymbol{R} \neq \boldsymbol{I}_I$, the content of the hidden activations might still be altered from time step to time step (although their norm remains constant). Consequently, the same information may be presented differently dependeing on for how many time steps it resided in memory. With $\boldsymbol{R} = \boldsymbol{I}_I$, it makes no difference for how long a piece of information had been stored, its representation will always be the same.

At this point, a practical example of the fully recurrent network could demonstrate the vanishing and exploding gradient problem. Possible examples:

- training a naive RNN on a simple task to memorize the first sequence element followed by a longer sequence of noise (similar to the latching problem in Bengio et al. (1994))
- plotting the determinant of $\partial \boldsymbol{s}(t)/(\partial \boldsymbol{s}(t-l))$ vs growing l
- train a network with $f(x) = x$ and $\boldsymbol{R}^\top \boldsymbol{R} = \boldsymbol{I}_I$! if it solves the simple long-term task, make it more difficult such that a linear model cannot solve it anymore.

@Freddy: Orthogonal matrices arise from the Euclidean norm assumption. If we go from Euclidean to ℓ_1 norm, do we end up in a stochastic matrix $\boldsymbol{R}$? No! It doesn't work on the level of normed spaces. Positivity of mass is essential here. This is a measure. So maybe a measure theoretic definition makes sense.

2.5.3 Focused backpropagation

Some of the problems described in the previous section were already addressed by Mozer (1989), who considered an architecture of the form

$$\begin{aligned} \boldsymbol{s}(t) &= \boldsymbol{W}^\top \boldsymbol{x}(t) \\ \boldsymbol{c}(t) &= f(\boldsymbol{s}(t)) + \boldsymbol{D}\, \boldsymbol{c}(t-1) \end{aligned} \tag{2.48}$$

where $\boldsymbol{D} \in \mathbb{R}^{I \times I}$ is a diagonal matrix. This means that the *context units* $\boldsymbol{c}(t)$ to not exchange information from one time step to another. The diagonal entries d_i of $\boldsymbol{D}$ may be considered as *decay factors* discounting stored information over time if smaller than 1. The diagonal structure of the recurrent weight matrix is similar to the Elman network but also to LSTM as we will see in the next chapter, the difference being that both, Elman and LSTM networks use identity, i.e. constant ones in the diagonal, while d_i are learned factors. The crucial property of this architecture, however, is that the context units integrate linearly over time. For simplicity, assume that only the last time step has a label, i.e. $L = L(\boldsymbol{y}(T), \hat{\boldsymbol{y}}(T))$, and consider the deltas

$$\boldsymbol{\delta}(t)^\top = \frac{\partial L}{\partial \boldsymbol{c}(t)} = \frac{\partial L}{\partial \boldsymbol{c}(t+1)} \frac{\partial \boldsymbol{c}(t+1)}{\partial \boldsymbol{c}(t)} = \boldsymbol{\delta}(t+1)^\top \boldsymbol{D}\,, \tag{2.49}$$

which can be equivalently computed as

$$\boldsymbol{\delta}(t)^\top = \frac{\partial L}{\partial \boldsymbol{c}(T)} \frac{\boldsymbol{c}(T)}{\boldsymbol{c}(t)} = \frac{\partial L}{\partial \boldsymbol{c}(T)} \frac{\boldsymbol{c}(T)}{\boldsymbol{c}(T-1)} \cdots \frac{\boldsymbol{c}(t+1)}{\boldsymbol{c}(t)} = \boldsymbol{\delta}(T)^\top \boldsymbol{D}^{T-t}. \tag{2.50}$$

Comparing Equation (2.50) to Equation (2.39) we find that in focused backpropagation the derivatives of the activation function do not multiply in each time step. By utilizing linear integration over time, focused backpropagation avoids this source of instability while learning.

We have seen that orthogonal matrices are norm preserving (cf. Equation (2.47)). If we require this property for the recurrent weight matrix in the focused architecture, we have to choose $\boldsymbol{D} = \boldsymbol{I}_I$. This is one of the central ideas of LSTM.

Chapter 3

Long Short-Term Memory

As we have seen in Section 2.5.2, the conditions for stable gradients over long distances are that the activation function of the recurrent units is the identity function $f(x) = x$ and that the recurrent weight matrix is orthogonal, i.e. $\boldsymbol{R}^\top \boldsymbol{R} = \boldsymbol{I}_I$. However, knowing these conditions does not solve the problem of long-term dependencies immediately as they lead to a very simple model with limited representative power.

Recall the self-recurrent unit that avoids vanishing gradients depicted in Figure 2.5. When we add an input to this unit as shown in Figure 3.1 all information from the input will be stored by the network indefinitely. Such a unit takes the form

$$c(t) = c(t-1) + z(t), \tag{3.1}$$

where $c(t)$ is the recurrent activation at time t and $z(t)$ is the input at time t. This unit stores all information provided by $z(t)$ indefinitely, which constitutes an inefficient use of the network capacity.

For this reason we add a gate $i(t) \in [0, 1] \subset \mathbb{R}$ to the unit's inlet that controls the input information stream. It can either be open (i.e. $i(t) = 1$), or closed (i.e. $i(t) = 0$), or something in between. A recurrent unit with input gate takes the form

$$c(t) = c(t-1) + z(t)i(t). \tag{3.2}$$

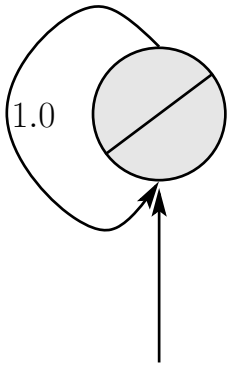


Figure 3.1: A single unit with self-recurrent connection which avoids the vanishing gradient and has an input.

Equation (3.2) still meets all criteria for stable gradients. During training the gate may learn to identify important information that is worth storing and will be useful for the current prediction or for later predictions. On the other hand it should learn to protect the internal memory from useless information or noise.

The long short-term memory (LSTM) is a recurrent neural network architecture that has *memory cells* as memory components. The LSTM architecture was first published in Hochreiter (1991) in German, then in a technical report (Hochreiter and Schmidhuber, 1995), in a workshop paper (Hochreiter and Schmidhuber, 1996), in a NeurIPS conference paper (Hochreiter and Schmidhuber, 1997b), and finally in the most cited paper (Hochreiter and Schmidhuber, 1997a). The memory cells of an LSTM network have gates that control the information flow into the memory and out of the memory. The input gate is completed by an *output gate* $o(t)$ which learns to access the memory of a memory cell. It protects jamming hidden units, other cells, itself, and the output layer with information that may be currently irrelevant but important later. A memory cell consists of

(i) g: the memory cell input activation function with activation z,

(ii) $i(t)$: the input gate with sigmoid activation function,

(iii) $c(t)$: the self-recurrent unit with $c(t) = c(t-1)+i(t)z(t)$, the identity as activation function, and self-recurrent weight 1.0,

(iv) $o(t)$: the output gate with sigmoid activation function,

(v) h: the memory cell activation function with cell state activation $h(c)$), and

(vi) $y(t)$: the memory cell activation with $y(t) = o(t) \odot h(c(t))$.

In vector notation (i.e. for multiple units), the forward rule is

$$\boldsymbol{i}(t) = \sigma(\boldsymbol{W}_i^\top \boldsymbol{x}(t) + \boldsymbol{R}_i^\top \boldsymbol{y}(t-1)) \tag{3.3}$$
$$\boldsymbol{o}(t) = \sigma(\boldsymbol{W}_o^\top \boldsymbol{x}(t) + \boldsymbol{R}_o^\top \boldsymbol{y}(t-1)) \tag{3.4}$$
$$\boldsymbol{z}(t) = g(\boldsymbol{W}_z^\top \boldsymbol{x}(t) + \boldsymbol{R}_z^\top \boldsymbol{y}(t-1)) \tag{3.5}$$
$$\boldsymbol{c}(t) = \boldsymbol{c}(t-1) + \boldsymbol{i}(t) \odot \boldsymbol{z}(t) \tag{3.6}$$
$$\boldsymbol{y}(t) = \boldsymbol{o}(t) \odot h(\boldsymbol{c}(t)), \tag{3.7}$$

where $\boldsymbol{i}(t), \boldsymbol{o}(t), \boldsymbol{z}(t), \boldsymbol{c}(t), \boldsymbol{y}(t) \in \mathbb{R}^I$. The vector $\boldsymbol{x}(t) \in \mathbb{R}^D$ is the *external input* and $\boldsymbol{y}(t-1)$ are the *memory cell activations* of the last time step. The vector $\boldsymbol{i}(t)$ is the *input gate activation*, $\boldsymbol{o}(t)$ is the *output gate activation*, $\boldsymbol{z}(t)$ is the *cell input activation*, $\boldsymbol{c}(t)$ is the *cell state*, and $\boldsymbol{y}(t)$ is the current memory cell activation vector. The function g is the *cell input activation function* and the function h is the *memory cell activation function*. $h(c)$ is the memory cell state activation. The operator $\odot$ is Hadamard's product, that is a point-wise (or element-wise) vector product.

Note: Notation

So far, we have used $\boldsymbol{y}(t)$ to denote the label at time t. In this chapter, however, we use $\boldsymbol{y}(t)$ to denote the memory cell activations. Further, we have used $g(\cdot)$ to denote the

model, whereas in this chapter it denotes the cell input activation function. In both cases it should become clear from the context which one is meant, otherwise it will be explicitly stated. We want to follow the standard notation for LSTM networks, therefore decided to overload the notation for y and g.

Note: Input Node

Hochreiter and Schmidhuber (1997a) used the sigmoid function $\sigma(\cdot)$ instead of the tangens hyperbolicus $\tanh(\cdot)$ for the input node (Equation 3.5).

Note: Original LSTM Connectivity

Hochreiter and Schmidhuber (1997a) used synaptic connections from $\boldsymbol{i}$, $\boldsymbol{o}$, $\boldsymbol{z}$, and $\boldsymbol{y}$ to all other units. For example gates could activate other gates and themselves.
Later that was reduced to connections from $\boldsymbol{y}$ to all other units.

The central element of the memory cell is Equation (3.6). The recurrent connections in this part of the network are fixed to the identity matrix (i.e. we have recurrent one-to-one connections), which is an orthogonal matrix as required to avoid vanishing or exploding gradients. Also, $\boldsymbol{c}(t)$ depends linearly on $\boldsymbol{c}(t-1)$, i.e. we have identity as activation function. As a consequence, the gradient is stable along this path through the time dimension. For this reason, this part is called the *constant error carousel* (CEC) since it traps the error signal and distributes it equally among the previous time steps (instead of exponentially decaying backwards in time).

Generally, a neural network can be interpreted as the weights being a long-term memory and the activations being a short-term memory. The name "Long Short-Term Memory" indicates that the short-term memory of this architecture, that is the cell state activations $\boldsymbol{c}(t)$, is capable of storing information over long distances in the input sequence.

The gates built around the CEC provide the model with capacity and protect the memory cell from distracting information. The gating mechanism makes LSTMs highly nonlinear.

Equation (3.5) describes the *cell input*. The activation function $g(\cdot)$ is most commonly chosen to be $\tanh(\cdot)$, which ensures that the activation of a cell input unit $z_i(t)$ is in the range $(-1, 1) \subset \mathbb{R}$. The choice of this activation is crucial. Using any of the most common activation functions like ReLU, logistic sigmoid, softplus, etc. in this place will usually deteriorate learning behavior significantly. The reason is that $\mathbb{E}(\boldsymbol{z}(t)) = 0$ is a desirable property, i.e. the expected value of the cell input activation should be zero (e.g. under the assumption of zero-mean Gaussian pre-activations). Otherwise, the cell state will quickly take on large values causing numerical problems. This is known as the *drift effect*, which is severe for sigmoid cell input activations. Note that in their original paper, Hochreiter and Schmidhuber (1997a) used the sigmoid function $g(\cdot) = \sigma(\cdot)$ as cell input activation. This variant will be discussed later in more detail.

The choice for the memory cell activation function $h(\cdot) = \tanh(\cdot)$ in Equation (3.7), on the other hand, is less critical. Also other functions like e.g. ReLU, SELU, softplus, etc. work quite well for many tasks.

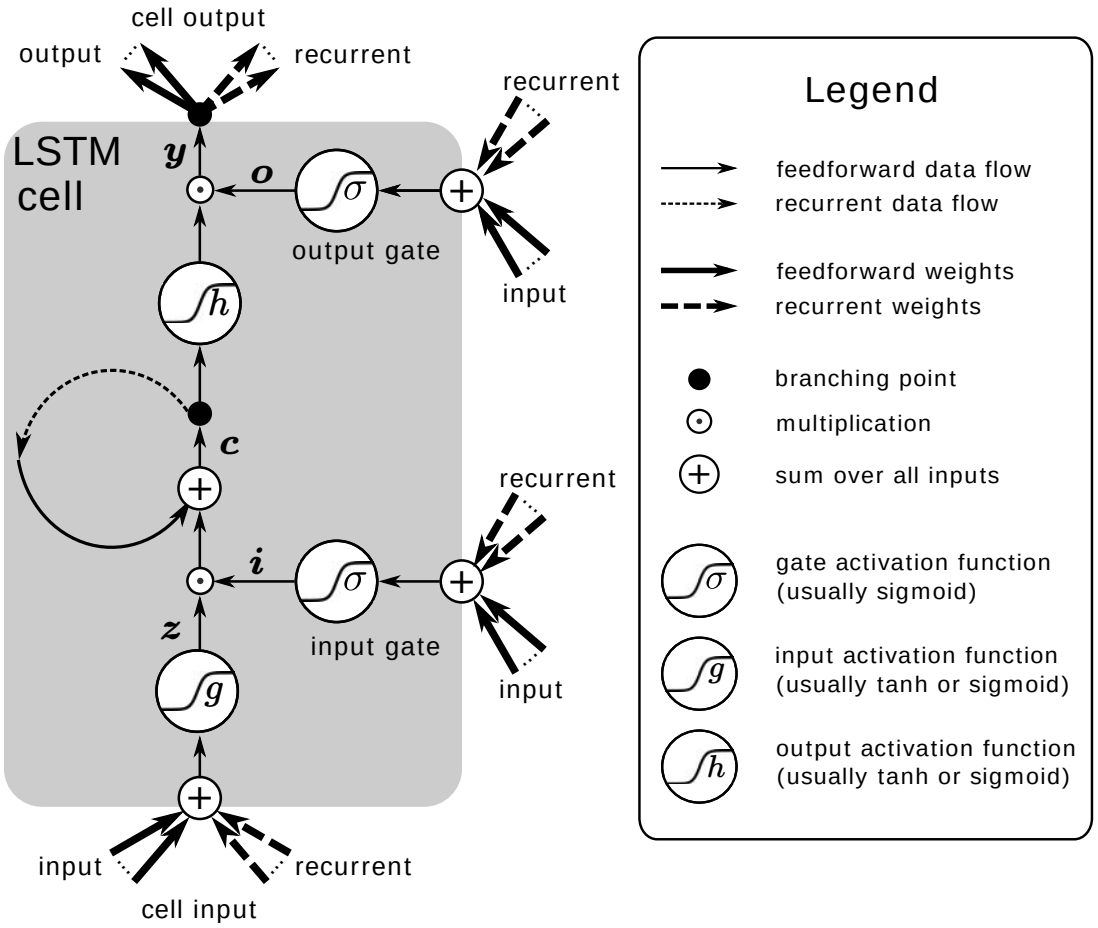


Figure 3.2: The LSTM memory cell. The memory cell input $\boldsymbol{z}(t)$ is often activated by $\tanh$. Subsequently, $\boldsymbol{z}(t)$ is multiplied by an input gate activation $\boldsymbol{i}(t)$. This product is added to the cell state $\boldsymbol{c}(t)$. The memory cell output $\boldsymbol{y}(t)$ is obtained by multiplying the memory cell state activation $h(\boldsymbol{c}(t))$ by an output gate activation $\boldsymbol{o}(t)$. Typically the memory cell activation function h is $\tanh$. The memory cell outputs are then recurrently fed into the cell input and the gates at the next time step $t + 1$ alongside with the new external input $\boldsymbol{x}(t + 1)$. The memory cell state $\boldsymbol{c}(t)$ carries error signals backwards in time without vanishing or exploding gradients.

Equations (3.3,3.4) constitute input gate and output gate, respectively. We define $\sigma(x) = (1 + e^{-x})^{-1}$ to be the logistic sigmoid function, i.e. an activation of a gate unit is in the interval $(0, 1) \subset \mathbb{R}$. Multiplying these gate activations to the cell input activation or cell state activation respectively can be thought of as gating these values in the sense that a gate activation of 0 corresponds to a closed gate (no information is allowed in) and 1 corresponds to an open gate (all information is allowed in). Therefore, the gates should learn to open up for important information and close for irrelevant information or noise.

3.1 Backpropagation for LSTM

In principle, all gradient-based learning methods discussed so far are applicable for LSTM training. Originally, LSTM was proposed to be trained using a combination of RTRL and truncated backpropagation, but nowadays the most common way to train LSTM networks is to use standard BPTT.

In the following, we will derive the gradients for LSTM. First, we denote the derivatives of the activation functions

$$\begin{aligned} \sigma(x) &= \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x} \\ \sigma'(x) &= \frac{e^x(1+e^x) - e^x e^x}{(1+e^x)^2} = \frac{e^x}{1+e^x}\frac{1}{1+e^x} = \sigma(x)(1-\sigma(x)) \end{aligned} \tag{3.8}$$

$$\begin{aligned} \tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \\ \tanh'(x) &= \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2} = 1 - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2} = 1 - \tanh^2(x), \end{aligned} \tag{3.9}$$

which are acquired using the quotient rule. In the following, let $L(t)$ be shorthand for $L(\boldsymbol{y}(t), \hat{\boldsymbol{y}}(t))$ and let L be shorthand for $L(\boldsymbol{y}(1{:}T), \hat{\boldsymbol{y}}(1{:}T))$. The gradient with respect to the cell activations $\boldsymbol{y}(t)$ is

$$\frac{\partial L}{\partial \boldsymbol{y}(t)} = \frac{\partial L(t)}{\partial \boldsymbol{y}(t)} + \frac{\partial L}{\partial \boldsymbol{i}(t+1)}\frac{\partial \boldsymbol{i}(t+1)}{\partial \boldsymbol{y}(t)} + \frac{\partial L}{\partial \boldsymbol{o}(t+1)}\frac{\partial \boldsymbol{o}(t+1)}{\partial \boldsymbol{y}(t)} + \frac{\partial L}{\partial \boldsymbol{z}(t+1)}\frac{\partial \boldsymbol{z}(t+1)}{\partial \boldsymbol{y}(t)} \tag{3.10}$$

where the first term $\partial L(t)/\partial \boldsymbol{y}(t)$ represents the error coming from the output layer at time t. The terms depending on $\boldsymbol{i}(t+1)$ and $\boldsymbol{o}(t+1)$ respectively account for the influence of $\boldsymbol{y}(t)$ on the gate activations at time $t+1$ given by the recurrent connections of the memory cell output. The term depending on $\boldsymbol{z}(t+1)$ accounts for the influence of $\boldsymbol{y}(t)$ on the memory cell input at time $t+1$ given by the recurrent connections of the memory cell output. In practice the error terms containing $\boldsymbol{i}(t+1)$, $\boldsymbol{o}(t+1)$, and $\boldsymbol{z}(t+1)$ are often omitted (i.e. set to zero) since the error through the CEC dominates the δ-errors of the backpropagation algorithm. δ-errors that flow through the gates will suffer from vanishing gradients. However, in some cases these error terms might have

large influence and even might lead to exploding gradients. These terms are:

$$\frac{\partial \boldsymbol{z}(t+1)}{\partial \boldsymbol{y}(t)} = \operatorname{diag}(g'(\boldsymbol{R}_z^\top \boldsymbol{y}(t) + \boldsymbol{W}_z^\top \boldsymbol{x}(t+1)))\boldsymbol{R}_z^\top \tag{3.11}$$

$$\frac{\partial \boldsymbol{i}(t+1)}{\partial \boldsymbol{y}(t)} = \operatorname{diag}(\sigma'(\boldsymbol{R}_i^\top \boldsymbol{y}(t) + \boldsymbol{W}_i^\top \boldsymbol{x}(t+1)))\boldsymbol{R}_i^\top \tag{3.12}$$

$$\frac{\partial \boldsymbol{o}(t+1)}{\partial \boldsymbol{y}(t)} = \operatorname{diag}(\sigma'(\boldsymbol{R}_o^\top \boldsymbol{y}(t) + \boldsymbol{W}_o^\top \boldsymbol{x}(t+1)))\boldsymbol{R}_o^\top \ . \tag{3.13}$$

Next, we consider the gradient w.r.t. the output gate, that is

$$\frac{\partial L}{\partial \boldsymbol{o}(t)} = \frac{\partial L}{\partial \boldsymbol{y}(t)} \frac{\partial \boldsymbol{y}(t)}{\partial \boldsymbol{o}(t)} = \frac{\partial L}{\partial \boldsymbol{y}(t)} \operatorname{diag}(h(\boldsymbol{c}(t))), \tag{3.14}$$

where $\partial L / \partial \boldsymbol{y}(t)$ was given in (3.10). The gradient with respect to the memory cell is

$$\begin{aligned} \frac{\partial L}{\partial \boldsymbol{c}(t)} &= \frac{\partial L}{\partial \boldsymbol{y}(t)} \frac{\partial \boldsymbol{y}(t)}{\partial \boldsymbol{c}(t)} + \frac{\partial L}{\partial \boldsymbol{c}(t+1)} \frac{\partial \boldsymbol{c}(t+1)}{\partial \boldsymbol{c}(t)} \\ &= \frac{\partial L}{\partial \boldsymbol{y}(t)} \operatorname{diag}(\boldsymbol{o}(t) \odot h'(\boldsymbol{c}(t))) + \frac{\partial L}{\partial \boldsymbol{c}(t+1)}. \end{aligned} \tag{3.15}$$

Equation (3.15) recursively sums up all error signals from the future and carries them backwards in time. Numerical stability over many recursion steps is the key to learning long-term dependencies. Finally, the gradients w.r.t. input gate $\boldsymbol{i}(t)$ and cell input $\boldsymbol{z}(t)$ are

$$\begin{aligned} \frac{\partial L}{\partial \boldsymbol{i}(t)} &= \frac{\partial L}{\partial \boldsymbol{c}(t)} \frac{\partial \boldsymbol{c}(t)}{\partial \boldsymbol{i}(t)} = \frac{\partial L}{\partial \boldsymbol{c}(t)} \operatorname{diag}(\boldsymbol{z}(t)) \\ \frac{\partial L}{\partial \boldsymbol{z}(t)} &= \frac{\partial L}{\partial \boldsymbol{c}(t)} \frac{\partial \boldsymbol{c}(t)}{\partial \boldsymbol{z}(t)} = \frac{\partial L}{\partial \boldsymbol{c}(t)} \operatorname{diag}(\boldsymbol{i}(t)), \end{aligned} \tag{3.16}$$

which completes the set of gradients w.r.t. activations, that is $\partial L / \partial \boldsymbol{\alpha}(t), \boldsymbol{\alpha} \in \{\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{z}, \boldsymbol{c}, \boldsymbol{y}\}$. Next, we define the deltas $\boldsymbol{\delta}_\alpha(t)$ as gradient w.r.t. the pre-activations of $\boldsymbol{\alpha}(t)$, where $\boldsymbol{\alpha} \in \{\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{z}\}$, i.e.

$$\begin{aligned} \boldsymbol{\delta}_i(t)^\top &= \frac{\partial L}{\partial \boldsymbol{i}(t)} \frac{\partial \boldsymbol{i}(t)}{\partial \left(\boldsymbol{W}_i^\top \boldsymbol{x}(t) + \boldsymbol{R}_i^\top \boldsymbol{y}(t-1)\right)} \\ &= \frac{\partial L}{\partial \boldsymbol{i}(t)} \operatorname{diag}(\sigma'(\boldsymbol{W}_i^\top \boldsymbol{x}(t) + \boldsymbol{R}_i^\top \boldsymbol{y}(t-1))) \end{aligned} \tag{3.17}$$

$$\begin{aligned} \boldsymbol{\delta}_o(t)^\top &= \frac{\partial L}{\partial \boldsymbol{o}(t)} \frac{\partial \boldsymbol{o}(t)}{\partial \left(\boldsymbol{W}_o^\top \boldsymbol{x}(t) + \boldsymbol{R}_o^\top \boldsymbol{y}(t-1)\right)} \\ &= \frac{\partial L}{\partial \boldsymbol{o}(t)} \operatorname{diag}(\sigma'(\boldsymbol{W}_o^\top \boldsymbol{x}(t) + \boldsymbol{R}_o^\top \boldsymbol{y}(t-1))) \end{aligned} \tag{3.18}$$

$$\begin{aligned} \boldsymbol{\delta}_z(t)^\top &= \frac{\partial L}{\partial \boldsymbol{z}(t)} \frac{\partial \boldsymbol{z}(t)}{\partial \left(\boldsymbol{W}_z^\top \boldsymbol{x}(t) + \boldsymbol{R}_z^\top \boldsymbol{y}(t-1)\right)} \\ &= \frac{\partial L}{\partial \boldsymbol{z}(t)} \operatorname{diag}(g'(\boldsymbol{W}_z^\top \boldsymbol{x}(t) + \boldsymbol{R}_z^\top \boldsymbol{y}(t-1))). \end{aligned} \tag{3.19}$$

Note that we defined $\boldsymbol{\delta}_\alpha(t)$ as column vectors since $\partial L / (\partial \boldsymbol{\alpha}(t))$ is a row vector by convention. With these deltas we can write the gradients for the whole weight matrices as sums of outer vector

products as

$$\frac{\partial L}{\partial \boldsymbol{R}_i} = \sum_{t=1}^{T} \boldsymbol{\delta}_i(t)\boldsymbol{y}(t-1)^\top \qquad \frac{\partial L}{\partial \boldsymbol{W}_i} = \sum_{t=1}^{T} \boldsymbol{\delta}_i(t)\boldsymbol{x}(t)^\top \tag{3.20}$$

$$\frac{\partial L}{\partial \boldsymbol{R}_o} = \sum_{t=1}^{T} \boldsymbol{\delta}_o(t)\boldsymbol{y}(t-1)^\top \qquad \frac{\partial L}{\partial \boldsymbol{W}_o} = \sum_{t=1}^{T} \boldsymbol{\delta}_o(t)\boldsymbol{x}(t)^\top \tag{3.21}$$

$$\frac{\partial L}{\partial \boldsymbol{R}_z} = \sum_{t=1}^{T} \boldsymbol{\delta}_z(t)\boldsymbol{y}(t-1)^\top \qquad \frac{\partial L}{\partial \boldsymbol{W}_z} = \sum_{t=1}^{T} \boldsymbol{\delta}_z(t)\boldsymbol{x}(t)^\top. \tag{3.22}$$

3.1.1 The LSTM learning method

Hochreiter and Schmidhuber (1997a) originally introduced LSTM alongside with a special training procedure, the *LSTM learning method*. It avoids exploding and vanishing gradients by truncating the δ-propagation through time over all units except the memory state $\boldsymbol{c}$. δ-errors arrived at units which are gates and memory cell input but are not further propagated back. Therefore, only the weights to these units are adjusted but not subsequent weights in the backpropagation procedure. For the training algorithm, this means that Equation (3.10) simplifies to

$$\frac{\partial L}{\partial \boldsymbol{y}(t)} = \frac{\partial L(t)}{\partial \boldsymbol{y}(t)}. \tag{3.23}$$

Another advantage of the LSTM learning method is that gates are used as gates and are not misused by other units like other gates for information processing.

3.2 Forget gate

Although the LSTM solves the vanishing gradient problem, there can still be cases where the network will struggle to learn. A well known reason for this is that for very long input sequences the cell states might grow indefinitely. One can think of this behavior as the network being unable to forget anything it has stored so far during processing of a sequence. That is to say, the LSTM cannot free memory resources that are no longer needed. Over time all memory cells are used up.

As a countermeasure Gers et al. (2000) proposed to add a third gate to the architecture, which is capable of resetting cell states when needed. The gate is analogous to input and output gate, i.e.

$$\boldsymbol{f}(t) = \sigma(\boldsymbol{W}_f^\top \boldsymbol{x}(t) + \boldsymbol{R}_f^\top \boldsymbol{y}(t-1)), \tag{3.24}$$

and used it within Equation (3.6) as:

$$\boldsymbol{c}(t) = \boldsymbol{f}(t) \odot \boldsymbol{c}(t-1) + \boldsymbol{i}(t) \odot \boldsymbol{z}(t). \tag{3.25}$$

This adaption enables the LSTM to delete content from the cell states when needed. However, the forget gate affects the gradient (3.15) in a peculiar way. Without forget gate we have

$$\frac{\partial \boldsymbol{c}(t+1)}{\partial \boldsymbol{c}(t)} = \boldsymbol{I}_I\ , \tag{3.26}$$

which is essential for avoiding the vanishing gradient problem. With forget gate, however, this gradient becomes

$$\frac{\partial \boldsymbol{c}(t+1)}{\partial \boldsymbol{c}(t)} = \operatorname{diag}(\boldsymbol{f}(t+1)) \,, \tag{3.27}$$

which is generally not an orthogonal matrix anymore, i.e. it violates condition (2.47). Consequently, the CEC gradient (3.15) becomes

$$\frac{\partial L}{\partial \boldsymbol{c}(t)} = \frac{\partial L}{\partial \boldsymbol{y}(t)} \operatorname{diag}(\boldsymbol{o}(t) \odot h(\boldsymbol{c}(t))) + \frac{\partial L}{\partial \boldsymbol{c}(t+1)} \operatorname{diag}(\boldsymbol{f}(t+1)) \,. \tag{3.28}$$

The forget gate may have a problematic gradient for long sequences since it is used multiple times. For example a small change of the bias weight of the forget gate can have large consequences for long sequences.

To mitigate this problem it is often advisable to initialize the forget gate bias units with large positive values. This biases the forget gate towards being closed (having activations close to one) and the gradient (3.27) gets closer to the identity matrix. Further, it should be noted that using a forget gate causes the network to put more emphasis on recent inputs. This may be desirable for some tasks like predicting the next character in a text corpus, where the prediction depends largely on local context. For other tasks, however, say predicting the winner of a chess game from a sequence of moves (where the model does not see the board but must keep track of the game on its own) the key patterns might be found in early stages of the game, that is in the beginning of the sequence. In such settings, the forget gate might *introduce* difficulties for learning the task.

Note: A Popular LSTM Version: Vanilla LSTM

The introduction of the forget gate led to one of the most used forms of the LSTM, which is now routinely used in many standard implementations and libraries. This version of the LSTM is defined by the following set of equations :

$$\begin{aligned}
\boldsymbol{i}(t) &= \sigma(\boldsymbol{W}_i^\top \boldsymbol{x}(t) + \boldsymbol{R}_i^\top \boldsymbol{y}(t-1)) \\
\boldsymbol{o}(t) &= \sigma(\boldsymbol{W}_o^\top \boldsymbol{x}(t) + \boldsymbol{R}_o^\top \boldsymbol{y}(t-1)) \\
\boldsymbol{f}(t) &= \sigma(\boldsymbol{W}_f^\top \boldsymbol{x}(t) + \boldsymbol{R}_f^\top \boldsymbol{y}(t-1)) \\
\boldsymbol{z}(t) &= g(\boldsymbol{W}_z^\top \boldsymbol{x}(t) + \boldsymbol{R}_z^\top \boldsymbol{y}(t-1)) \\
\boldsymbol{c}(t) &= \boldsymbol{f}(t) \odot \boldsymbol{c}(t-1) + \boldsymbol{i}(t) \odot \boldsymbol{z}(t) \\
\boldsymbol{y}(t) &= \boldsymbol{o}(t) \odot h(\boldsymbol{c}(t)).
\end{aligned} \tag{3.29}$$

This LSTM variant is depicted in Figure 3.3.

An often used variation of the forget gate is to tie it to the input gate (Jaeger, 2002). This removes the necessity of introducing additional parameters in that Equation (3.25) changes to

$$\boldsymbol{c}(t) = (\boldsymbol{1} - \boldsymbol{i}(t)) \odot \boldsymbol{c}(t-1) + \boldsymbol{i}(t) \odot \boldsymbol{z}(t), \tag{3.30}$$

i.e. a cell deletes old information only to the extent to which new information is allowed in. Tallec and Ollivier (2018a) argue that an RNN of a form like (3.30) arises naturally if one requires the network to be robust against time warps. The argument interprets Equation (3.1) as a discretization

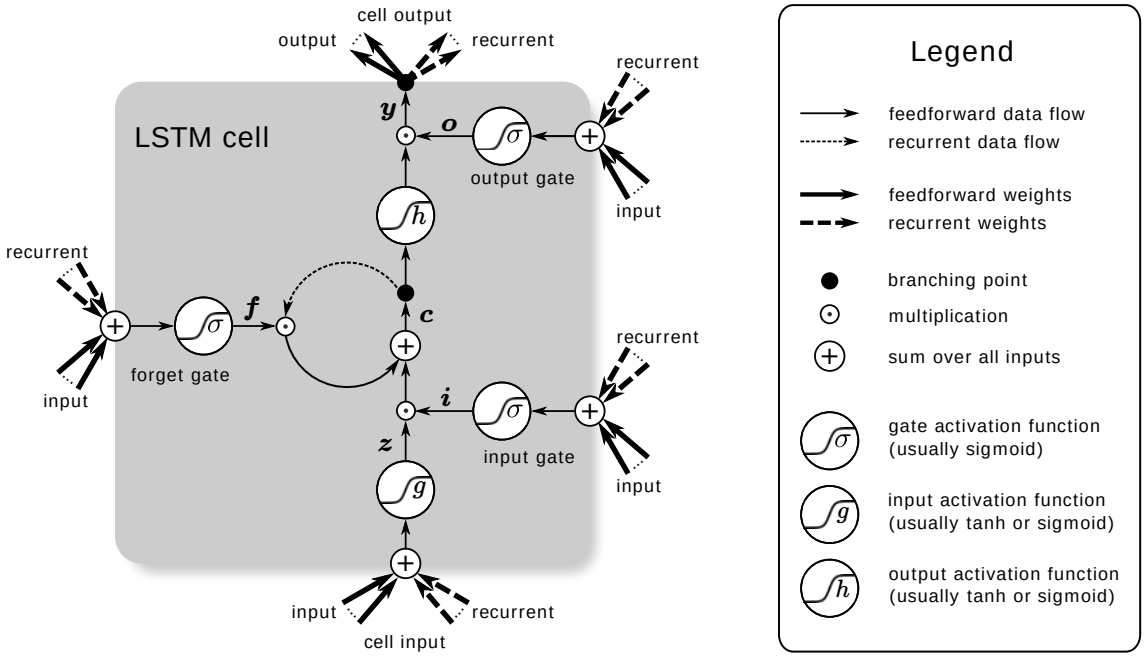


Figure 3.3: Vanilla LSTM. An additional forget gate $\boldsymbol{f}(t)$ can reset the memory cell which allows the system to "forget". This variant is implemented in most deep learning frameworks.

of a continuous-time differential equation and then substitutes the time parameter t by $\omega(t)$, where $\omega : \mathbb{R}_+ \to \mathbb{R}_+$ is some warping function. When transforming the equation back into discrete time the result takes the form of (3.30), where $\boldsymbol{i}(t)$ takes the role of the warping function (in this form each cell can be warped differently). mention chrono init here?

In 2014 this idea was proposed as the *gated recurrent unit* (GRU) by Cho et al. as a simplification of the LSTM. GRUs are now famous alternatives to LSTM memory cells.

3.3 Tricks of the trade

Tricks of the trade are clever techniques that are often used by experts. In this chapter we explore a few of them. Many of them have never been published and are widely unknown.

Focused LSTM We describe the focused LSTM architecture which has no forget gate and fewer parameters than the vanilla LSTM. Its main characteristic is that it separates the input $\boldsymbol{x}(t)$ from previous cell states $\boldsymbol{y}(t-1)$ when activating the cell input and the gates. For a sigma-pi unit (a multiplicative unit) composed of two units, both units have to be active to activate the sigma-pi unit. Typically, either the input gate or the cell input activation is constant or both learn the same pattern. In these cases only one unit would be sufficient. In the reduced architecture the units can focus on either external information (cell input) or internal information (gates).

The gates are activated by previous cell states $\boldsymbol{y}(t-1)$ while the cell input is activated by the input $\boldsymbol{x}(t)$. Therefore the cell input detects patterns in the input while the input gate decides based on the current network state if the pattern is stored or not. Also access to the memory is decided

based on the current state of the network via the output gates. The definition of LSTM given in Equations (3.3-3.7) has many parameters, in fact $3I^2+3ID$ without bias units. We will now give a variant that uses only $2I^2 + ID$ parameters. The only way how external information can enter the LSTM network is the cell input $\boldsymbol{z}(t)$. With signed activations it decides whether the corresponding cell state will grow or shrink and the inputs $\boldsymbol{x}(t)$ are encoded here. The input gate $\boldsymbol{i}(t)$ can only scale the signal provided by $\boldsymbol{z}(i)$, i.e. it can decide on the importance of the information at a given time but it cannot change its content. Unlike $\boldsymbol{x}(t)$, the cell activations $\boldsymbol{y}(t)$ carry information about the whole sequence from time 0 up to t. The cell activations $\boldsymbol{y}(t)$ carry information about the position in the sequence and the relative time when processing the sequence. Absolute and relative time in sequences processing are often useful for controlling the gating mechanism. These considerations lead to a smaller variant, the focused LSTM, depicted in Figure 3.4 and defined by the equations

$$\begin{aligned}
\boldsymbol{i}(t) &= \sigma(\boldsymbol{R}_i^\top y(t-1)) \,, \\
\boldsymbol{o}(t) &= \sigma(\boldsymbol{R}_o^\top y(t-1)) \,, \\
\boldsymbol{z}(t) &= g(\boldsymbol{W}^\top \boldsymbol{x}(t)) \\
\boldsymbol{c}(t) &= \boldsymbol{c}(t-1) + \boldsymbol{i}(t) \odot \boldsymbol{z}(t) \,, \\
\boldsymbol{y}(t) &= \boldsymbol{o}(t) \odot h(\boldsymbol{c}(t)) \,.
\end{aligned} \tag{3.31}$$

The focused LSTM is a compact LSTM version, where the gates depend on the cell output $\boldsymbol{y}(t)$ and the cell input on $\boldsymbol{x}(t)$. The vanilla LSTM usually feeds both $\boldsymbol{y}(t)$ and $\boldsymbol{x}(t)$ into $\boldsymbol{i}(t), \boldsymbol{o}(t), \boldsymbol{z}(t)$, and $\boldsymbol{f}(t)$.

Lightweight LSTM The lightweight LSTM is the focused LSTM without output gates. For sigmoid input activation function it has the property that the memory content is always growing and will more and more activate or deactivate the output. Each memory cell collects hints for or against the output it is supporting by its weight to the output layer. The states are only changed by current input and input gate, therefore contributions to cell states are Markov: their influence onto other units cannot be changed in the future.

Ticker steps An important conception in deep learning is that a large number of layers enables heavy data abstraction and is considered key to its success. An RNN for sequence classification (i.e. the RNN only makes one prediction at the end of the sequence) can be interpreted as a neural network with its number of layers equal to the length of the input sequence. If one seeks to use a deeper net than naturally defined by the sequence length, one can add additional elements at the end of the sequence. We refer to these elements as *ticker steps*. This increases the depth of the network and can increase the level of abstraction learned by the network. The additional elements can either be neutral (i.e. zero vectors) or in some cases it might make sense to feed some special non-zero element repeatedly. Figure 3.6 demonstrates the concept of ticker steps at the example of semantic segmentation.

Gate biases When working with very long sequences it can happen that the cell states get very large values over time, i.e. the cell states are drifting. The *drift effect* hinders learning and can lead to numerical difficulties. To prevent the drift effect, the input gate can be initialized with a negative

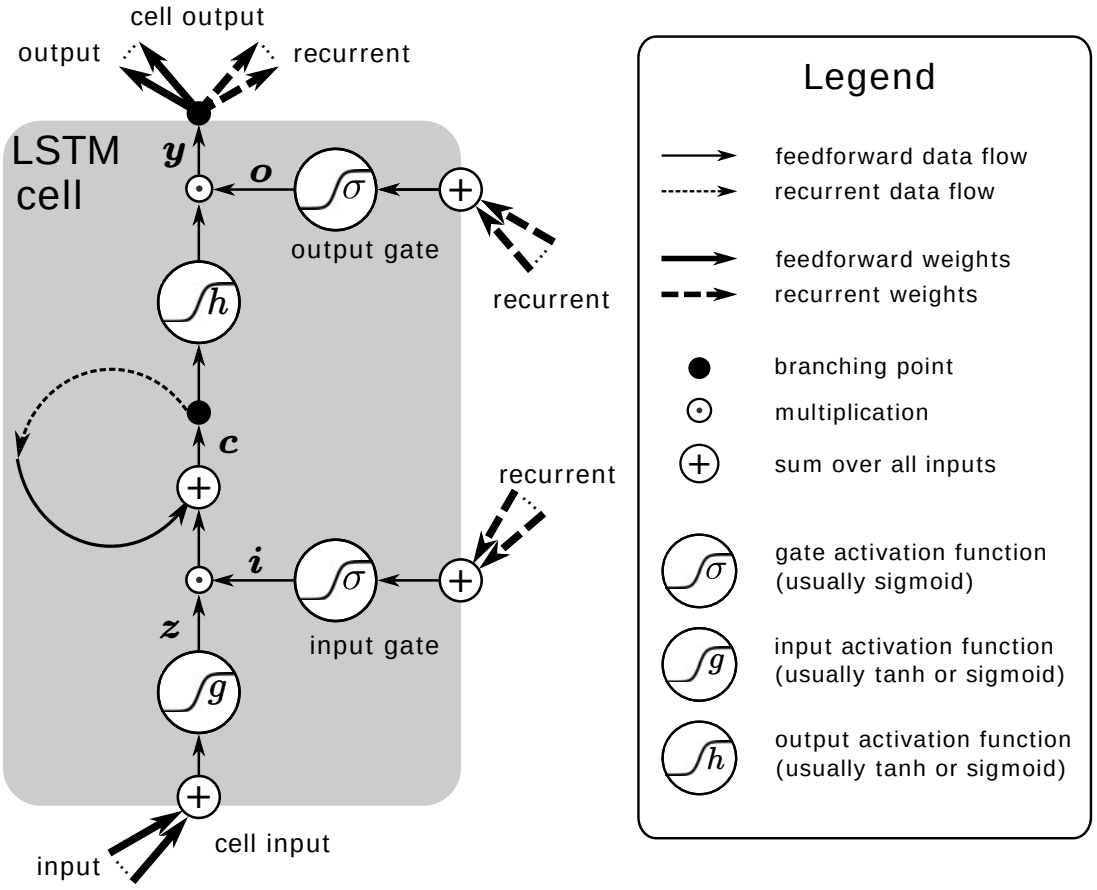


Figure 3.4: Focused LSTM. The cell outputs $\boldsymbol{y}(t)$ are recurrently connected only to the gates $\boldsymbol{i}(t) and \boldsymbol{o}(t)$ but not to the cell input $\boldsymbol{z}(t)$. On the other hand, the external inputs $\boldsymbol{x}(t)$ are only fed to the cell input but not to the gates. This saves about half of the parameters of the standard implementation. The cell input focuses on external information while the gates on internal information.

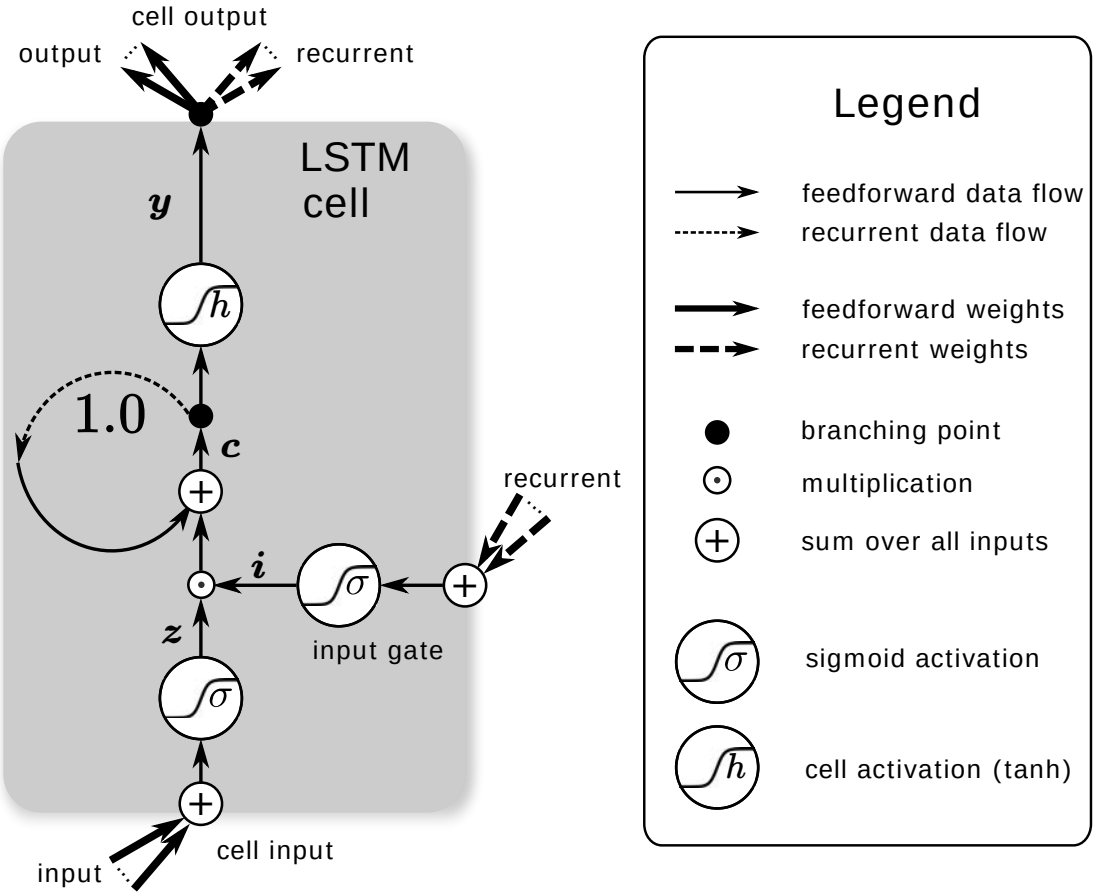


Figure 3.5: Lightweight LSTM is a focused LSTM without output gates. It has Markov properties concerning the the contribution to cell states.

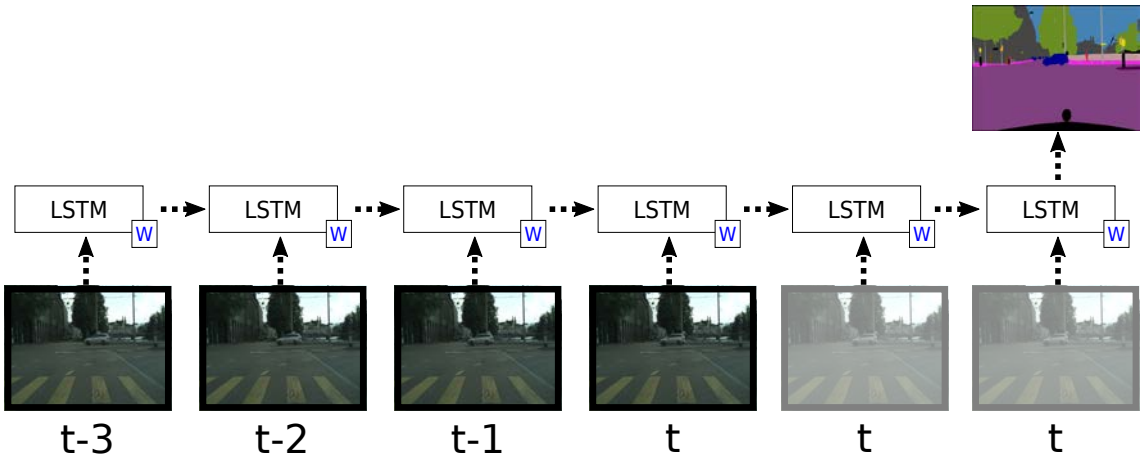


Figure 3.6: The task here is to predict a semantic map of the last image in the sequence. The LSTM processes the sequence to the last element at time t. To provide the network with more abstractive and representative power two ticker steps are added to the sequence before the actual prediction is made.

bias, e.g. $-1, -2, \ldots, -10$. This gears the LSTM's initial behavior towards ignoring most of its inputs and the network should learn to open the input gates only for relevant information. As a consequence, the cell state increments become smaller in expectation which prevents excessive accumulation.

Additionally, it can sometimes be helpful to encourage the network not to use all memory cells at the same time. If we initialize the input and output gate biases with a sequence, e.g. $((1-i)/2)_{i=1}^{I}$, where i enumerates the memory cells, then this introduces a ranking of the memory cells. The memory cell $i = 1$ has a bias of 0, i.e. the input gate activation is $1/2$ in expectation. As i grows the gate biases become more negative. This means that by default the first unit will have larger δ-errors than the second, the second unit will have larger δ-errors than the third, and so on. Consequently, the network will not use all units equally, but will first try to solve the task using only memory cells with lower indices. The use of units with higher indices must first be enabled by BPTT adjusting biases or weights accordingly. Equation (3.15) states that the output gate activation scales the error signal that flows into the cell states. That is, units with higher output gate bias will tend to carry a stronger learning signal than units with lower bias. Overall, this initialization scheme introduces a ranking among the memory cells that encourages the network to use as few memory cells as possible, thereby saving capacity for later learning stages. The memory cells kick in one by one during learning. This bias scheme was very successful in Hochreiter et al. (2007a) for protein sequence analysis.

Scaling of g or h. If one suspects that these long sequences contain only a few elements which are relevant for the task, then *input activation scaling* might be helpful in addition to negative biases. That is, instead of g one uses αg where $\alpha > 1$ is some scaling constant(e.g. $\alpha = 4$). The input then tends in the cases where the gate opens to provides a stronger signal. For example if there is only one pattern in the sequence, then the detection of this pattern should lead to a contribution to the memory state that is clearly recognizable and should activate the cell output.

If instead of h one uses αh where $\alpha > 1$, then also the cell state is amplified. This might be important if small changes in the memory content should be recognizable.

Linear g or h. In many applications a linear g – often slope one – performs better than using a non-linear activation function. The gradient of the activation function does not scale the gradient if the slope is one.

A linear h is of importance if some counting or accumulation signal is stored in the memory. In particular extrapolation works fine. For example see the RUDDER paper (Arjona-Medina et al., 2019) where rewards are still correctly predicted by an LSTM when the agent became more successful and got more reward.

Sigmoid vs. tanh for g. A sigmoid input activation function is advantageous for detecting single patterns that rarely appear in the input sequence. This was successful in recognizing patterns in protein sequences (Hochreiter et al., 2007a).

Tanh is important if hints for and against a fact that is indicated by the memory cell are found in the input sequence. For example in text analysis there are words that hint to a fact like sport-related text and words that hint in another direction. Also for adjusting parameters that are stored in the

memory cells tanh is helpful. For example when using LSTM for learning to learn (Hochreiter et al., 2001), it has to store the actual parameters.

Absolute timing In some tasks it can be beneficial for the network to have access to the time index t (i.e. use it as part of the input). For example, if the input data are weather time series ranging over a couple of years, then knowing the month in which a data point was measured might already constitute valuable information about the prediction task (simply, August in Austria tends to be warmer than December). However, if one simply feeds t as additional input variable to the LSTM it will accumulate these values in the cell state which is not the desired behavior. Instead, it is often better to feed periodic signals in parallel to the input sequence. For instance, one can feed two additional input variables $\sin(\alpha t), \cos(\alpha t)$, where the scaling constant $\alpha > 0$ can be used to adjust the time scale to the that of the input sequence. In this manner the network can naturally develop absolute time awareness during learning.

Most successful are Gaussian (radial basis functions) or triangle signals or other locality indicating functions that are distributed across the input sequence. The activation of a particular Gaussian tells how far the actual input is away from this Gaussian.

To distinguish few discrete steps, a binary time counter can be added. First counter distinguishes between even and odd time steps, the second is modulo 4, and the third counter is modulo 8.

Fully connected gates Hochreiter and Schmidhuber (1997a) used synaptic connections from $\boldsymbol{i}$, $\boldsymbol{o}$, $\boldsymbol{z}$, and $\boldsymbol{y}$ to all other units. For example gates could activate other gates and themselves. For this variant it is especially important to use the LSTM learning method (Hochreiter and Schmidhuber, 1997a) to avoid exploding and vanishing gradients through the gates.

The gates provide the LSTM architecture with its remarkable representative power. They learn to filter information, protect the memory cells from being jammed, and retrieve data at certain points in time, i.e. they learn complex temporal behavior. This behavior contains valuable information itself and sometimes it is beneficial to inform the gates about what the other gates are doing. Then the output gate of a cell could, e.g., learn to open every time its input gate closes. This means, the gate does not need to detect the corresponding pattern in the input data but directly from the behavior of the other gate.

Input gates are activated by other input gates and cell inputs which makes them sensitive for pattern if only a preceding pattern was present.

An LSTM network with fully connected gates can learn very complex dynamics. It is especially important for this very architecture to truncate the gradient behind the gates. The forward pass is, e.g.

$$\begin{aligned}
\boldsymbol{z}(t) &= g\left(\boldsymbol{W}_{xz}^\top \boldsymbol{x}(t) + \boldsymbol{W}_{zz}^\top \boldsymbol{z}(t-1) + \boldsymbol{W}_{iz}^\top \boldsymbol{i}(t-1) + \boldsymbol{W}_{oz}^\top \boldsymbol{o}(t-1) + \boldsymbol{W}_{yz}^\top \boldsymbol{y}(t-1)\right) \\
\boldsymbol{i}(t) &= \sigma\left(\boldsymbol{W}_{xi}^\top \boldsymbol{x}(t) + \boldsymbol{W}_{zi}^\top \boldsymbol{z}(t-1) + \boldsymbol{W}_{ii}^\top \boldsymbol{i}(t-1) + \boldsymbol{W}_{oi}^\top \boldsymbol{o}(t-1) + \boldsymbol{W}_{yi}^\top \boldsymbol{y}(t-1)\right) \\
\boldsymbol{o}(t) &= \sigma\left(\boldsymbol{W}_{xo}^\top \boldsymbol{x}(t) + \boldsymbol{W}_{zo}^\top \boldsymbol{z}(t-1) + \boldsymbol{W}_{io}^\top \boldsymbol{i}(t-1) + \boldsymbol{W}_{oo}^\top \boldsymbol{o}(t-1) + \boldsymbol{W}_{yo}^\top \boldsymbol{y}(t-1)\right) \\
\boldsymbol{c}(t) &= \boldsymbol{c}(t-1) + \boldsymbol{i}(t) \odot \boldsymbol{z}(t) \\
\boldsymbol{y}(t) &= \boldsymbol{o}(t) \odot h(\boldsymbol{c}(t)),
\end{aligned} \tag{3.32}$$

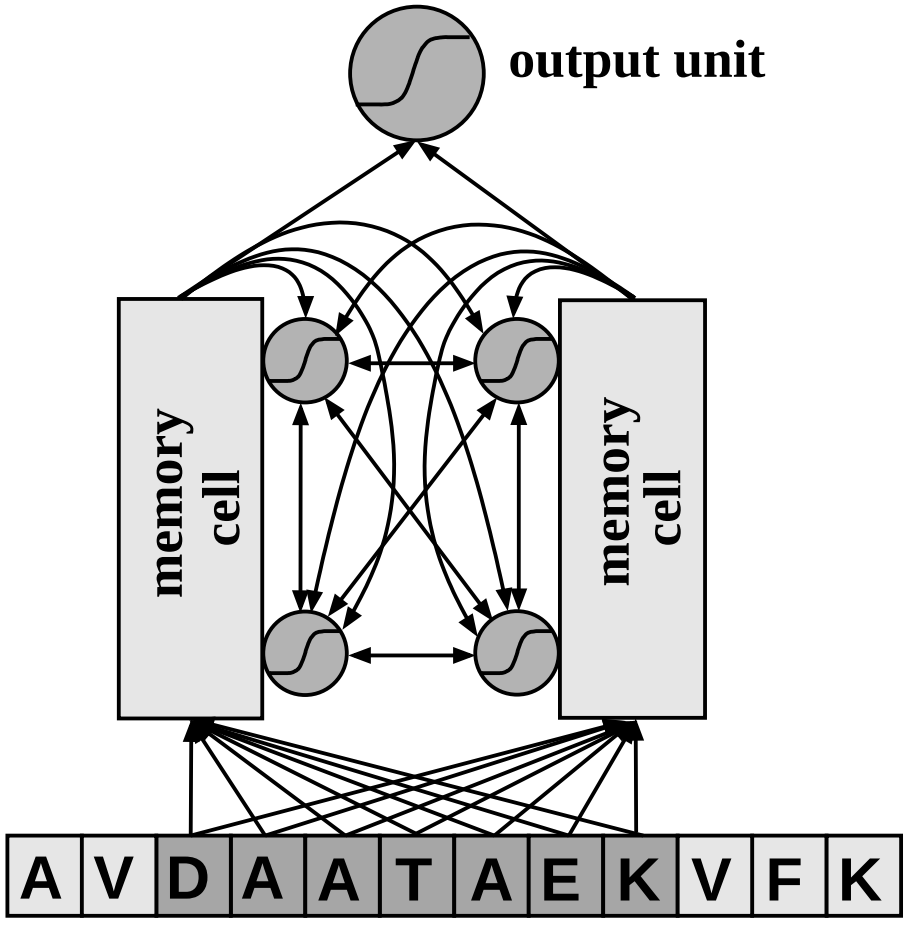


Figure 3.7: LSTM network with fully connected gates.

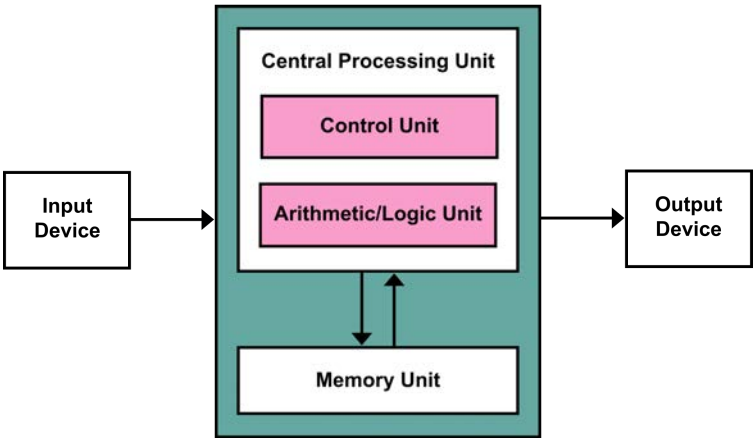


Figure 3.8: The von Neumann architecture. We can realize a neural version of this concept by implementing the memory unit by an LSTM and the central processing unit by a fully connected recurrent net. Source: `https://en.wikipedia.org/wiki/Von_Neumann_architecture`

but this is not the only possibility to wire the gates with each other. Also other variants are possible.

Separation of Memory and Compute We can provide the previous idea with some additional structure. Consider the von Neumann architecture that divides the responsibilities of a computer into two distinct units, a central processing unit that performs arithmetic and control, and a memory that stores data and instructions. Figure 3.8 depicts this concept.

The memory cells can store information over long periods of time and can therefore be used as memory unit. A fully connected recurrent layer has only a very short-term memory but can do complicated operations in this context. It can therefore be used as processing unit in the von Neumann architecture. These two components connect with each other such that information can flow from the memory to the processing unit (read access) and that computation results can be stored for later use (write access). Therefore, we connect the outputs $\boldsymbol{y}(t-1)$ of the memory cells at time $t-1$ to the input of the fully recurrent layer at time t and we connect the hidden activations $\boldsymbol{a}(t-1)$ of the fully recurrent layer at time $t-1$ with the cell and gate inputs at time t. This results in the forward rule

$$\begin{aligned}
\boldsymbol{i}(t) &= \sigma(\boldsymbol{W}_i^\top \boldsymbol{x}(t) + \boldsymbol{R}_i^\top \boldsymbol{y}(t-1) + \boldsymbol{U}_i^\top \boldsymbol{a}(t-1)) \\
\boldsymbol{o}(t) &= \sigma(\boldsymbol{W}_o^\top \boldsymbol{x}(t) + \boldsymbol{R}_o^\top \boldsymbol{y}(t-1) + \boldsymbol{U}_o^\top \boldsymbol{a}(t-1)) \\
\boldsymbol{z}(t) &= g(\boldsymbol{W}_z^\top \boldsymbol{x}(t) + \boldsymbol{R}_z^\top \boldsymbol{y}(t-1) + \boldsymbol{U}_z^\top \boldsymbol{a}(t-1)) \\
\boldsymbol{c}(t) &= \boldsymbol{c}(t-1) + \boldsymbol{i}(t) \odot \boldsymbol{z}(t) \\
\boldsymbol{a}(t) &= f(\boldsymbol{W}_a^\top \boldsymbol{x}(t) + \boldsymbol{R}_a^\top \boldsymbol{a}(t-1) + \boldsymbol{U}_a^\top \boldsymbol{y}(t-1)) \\
\boldsymbol{y}(t) &= \boldsymbol{o}(t) \odot h(\boldsymbol{c}(t)).
\end{aligned} \tag{3.33}$$

Finally, we can concatenate the output vectors $\boldsymbol{y}(t)$ and $\boldsymbol{a}(t)$ and feed them to an appropriate output layer. Memory cell is treated like a standard recurrent hidden unit. It is, however, difficult to train such an architecture properly because the fully recurrent layer learns much faster than the memory cells.

One way to deal with this is to put all controls into the hands of the fully recurrent layer. That is, inputs are fed to the fully recurrent layer only, and the memory cells operate on the hidden representations of the fully recurrent layer. Additionally, one can add a recurrent connection from the memory cell outputs to the input of the fully recurrent layer so as to provide the fully recurrent layer with read access to the memory. This results in the forward rule

$$\begin{aligned}
\boldsymbol{a}(t) &= f(\boldsymbol{W}_a^\top \boldsymbol{x}(t) + \boldsymbol{R}_a^\top \boldsymbol{a}(t-1) + \boldsymbol{U}_a^\top \boldsymbol{y}(t-1)) \\
\boldsymbol{i}(t) &= \sigma(\boldsymbol{W}_i^\top \boldsymbol{a}(t) + \boldsymbol{R}_i^\top \boldsymbol{y}(t-1)) \\
\boldsymbol{o}(t) &= \sigma(\boldsymbol{W}_o^\top \boldsymbol{a}(t) + \boldsymbol{R}_o^\top \boldsymbol{y}(t-1)) \\
\boldsymbol{z}(t) &= g(\boldsymbol{W}_z^\top \boldsymbol{a}(t) + \boldsymbol{R}_z^\top \boldsymbol{y}(t-1)) \\
\boldsymbol{c}(t) &= \boldsymbol{c}(t-1) + \boldsymbol{i}(t) \odot \boldsymbol{z}(t) \\
\boldsymbol{y}(t) &= \boldsymbol{o}(t) \odot h(\boldsymbol{c}(t)).
\end{aligned} \tag{3.34}$$

If we use $\boldsymbol{a}(t)$ as output of this LSTM network, then both input and output are wired to the fully recurrent layer only. This prevents that the memory cells and the fully recurrent units compete during learning. The main information flow is through the fully recurrent layer only. Therefore, this layer will learn to solve as much as possible on its own, without using memory cells. If, however, the data contains long-terms dependencies, then the only chance for the fully recurrent layer to detect these dependencies is via the memory cells. The network must learn to control the memory cells properly, i.e. store important information and retrieve it again when necessary.

Input window Often not the actual input is presented to the LSTM but a whole window of inputs. A window has the advantage that an LSTM can learn patterns in the input via its cell input activation. This was successfully used in Hochreiter et al. (2007b). An LSTM detected motifs in protein sequences, stored them and at sequence end mapped the protein sequences to functional or structural classes based on the stored motifs.

Online learning In many applications the LSTM is updated after each sequence and not for a batch of sequences. Online learning is advantageous at beginning of learning since the online update has an exploratory effect.

More cells than necessary LSTM learning is sped up if one uses more cells than are necessary. With more cells the chance increases that an input activation function has an initialization close to a relevant pattern. Then storing can be learned right away. Otherwise there is the hen-egg problem: if storing has been learned then the right pattern can be learned since there is gradient information — if the right pattern has been learned then storing it can be learned. If the input activation extracts some relevant information then this can be learned to be stored and at the same time the relevant information content can be maximized.

Learning rate schedule For LSTM networks the learning rate can be decreased over time. In some cases it is beneficial if the learning rate is increased again at certain times to encourage exploration. Learning can often be sped up if only the learning rate of bias weights is decreased after some time.

Sequence classification It is of advantageous to process a whole sequence and at the end give the target instead of continuously predicting the target. The LSTM net can collect all information and then process it.

Continuously predicting the target has high errors at time steps when the target is not predictable. This can hinder learning. See RUDDER Arjona-Medina et al. (2019) for more on this topic.

Continuous prediction vs. multiple LSTM networks use multiple LSTMs instead of continuous prediction. For continuous predictions the memory cells have to store information that are relevant for all predictions. A piece of information that is relevant for only one prediction is hardly stored. Therefore the LSTM net can store all relevant information for a prediction without making compromises.

Target and input scaling Scaling the target to $[0.2, 0.8]$ avoids that output units get stuck at saturated states and learning stalls.

Standardize input to zero mean and unit variance. If outliers are present, then apply $\tanh$ and standardize again. Do this until the outliers are gone.

LSTM3 architecture

LSTM4 architecture

LSTM5 architecture

3.4 Dropout for LSTM

One obvious way to apply dropout to LSTM networks is to unfold the network in time and then use dropout on the resulting feedforward structure. However, for sufficiently long sequences, this will disturb the function of all memory cells. Let p be the dropout probability, then the probability for a unit to "survive" until the end of the sequence is $(1-p)^T$, i.e. it decays exponentially with time. The result is that the network cannot learn properly because its internal memory keeps being cleared. This fact especially hinders learning of long-term dependencies because the probability for the relevant information to "survive" the necessary time span becomes exponentially small.

These problems called for special dropout techniques designed specifically for LSTM architecture. One very simple remedy is to just not apply dropout to recurrent connections but to input connections only (Zaremba et al., 2014). Let $\boldsymbol{d}(t) \sim \mathcal{B}((1-p)\mathbf{1}_D)$ be a random vector of length I whose entries are all $(1-p)$-Bernoulli distributed, where p is the dropout probability, then the

forward rule becomes

$$\begin{aligned}
\boldsymbol{i}(t) &= \sigma(\boldsymbol{W}_i^\top(\boldsymbol{d}(t) \odot \boldsymbol{x}(t)) + \boldsymbol{R}_i^\top \boldsymbol{y}(t-1)) \\
\boldsymbol{o}(t) &= \sigma(\boldsymbol{W}_o^\top(\boldsymbol{d}(t) \odot \boldsymbol{x}(t)) + \boldsymbol{R}_o^\top \boldsymbol{y}(t-1)) \\
\boldsymbol{z}(t) &= g(\boldsymbol{W}_z^\top(\boldsymbol{d}(t) \odot \boldsymbol{x}(t)) + \boldsymbol{R}_z^\top \boldsymbol{y}(t-1)) \\
\boldsymbol{c}(t) &= \boldsymbol{c}(t-1) + \boldsymbol{i}(t) \odot \boldsymbol{z}(t) \\
\boldsymbol{y}(t) &= \boldsymbol{o}(t) \odot h(\boldsymbol{c}(t)) \,.
\end{aligned} \tag{3.35}$$

However, this method circumvents the problem more than it solves it.

Another LSTM dropout method that is not limited in this sense is called *zoneout* (Krueger et al., 2017), whose basic idea is to drop out the updates to the memory cell instead of clearing its entire content. In this manner, dependencies spanning over a dropout event can still be learned as long as the dropped-out update did not carry the crucial piece of information. Let $\boldsymbol{d}(t) \sim \mathcal{B}((1-p)\mathbf{1}_I)$ similar as before, then zoneout uses $\boldsymbol{d}(t)$ as dropout mask by altering Equations (3.6) and (3.7) to

$$\boldsymbol{c}(t) = \boldsymbol{c}(t-1) + \boldsymbol{d}(t) \odot \boldsymbol{i}(t) \odot \boldsymbol{z}(t) \tag{3.36}$$

$$\boldsymbol{y}(t) = (\mathbf{1} - \boldsymbol{d}(t)) \odot \boldsymbol{y}(t-1) + \boldsymbol{d}(t) \odot \boldsymbol{o}(t) \odot h(\boldsymbol{c}(t)), \tag{3.37}$$

during learning. Of course, during inference we want to make use of the full power of our network and do not drop out any activations.

3.5 Application examples

3.5.1 Sequence-to-sequence learning with LSTM

Although text encoding at character level is technically more straightforward than at word level, the latter has the advantage that since the input to the network is more high-level it has the potential to solve more complicated tasks with the same capacity. This is obvious because a network operating at character level first has to learn the concept of a word before starting to solve the main task. However, encoding text at word level is more challenging because number of elements to encode explodes from 26 plus some special characters to tens of thousands of words in a vocabulary.

Note: Word embedding

In natural language processing (NLP) tasks like translation, text-to-speech synthesis, etc. it is more convenient to operate at word level rather than a character level. While the latter is technically easier to handle, it complicates the task significantly as the concept of words needs to be learned before the model can start to learn about their semantics.
Therefore, many NLP models operate on a word level. This, however, raises the question of how to represent words to the NLP models if not in the form of strings. A very old idea is to use a *word embedding*, that is to have a vector space that hosts the vocabulary of an entire language (Salton, 1962; Salton et al., 1975). This idea is appealing as this enables words to maintain certain relations to one another, like distance, orthogonality, or being

additive inverse (opposite).
One example for a word embedding is *latent semantic analysis* (Dumais, 2004). This is a count-based method that relies on the assumption that words of similar meaning will occur in similar text corpora. Therefore, a word is represented by the vector consisting of the number of occurrences in a selection of text corpora. Principal component analysis (PCA) is then used to reduce the number of dimensions.
Later, Bengio et al. (2003) proposed a neural word embedding based on a next-word-prediction task. That is, a neural network was trained to minimize the perplexity of the model for the next word in a text sample of a given length. The input layer of this model is a linear mapping from one-hot word vectors (i.e. the size of such a vector is the size of the vocabulary) to the embedding space, which is a much smaller vector space. After training, the linear mapping can be used as a table for looking up words in the embedding space.
Mikolov et al. (2013a) presented the `word2vec` toolkit that utilizes two basically similar but somewhat inverse ideas: *continuous bag-of-words* (CBOW) and *continuous skip-gram*. The objective of CBOW is to predict a word given its context, i.e. a certain number of words before and after an occurrence of the target word in the text corpus. The order of the words in the context window does not matter. Therefore, it is a "bag of words". The skip-gram model predicts the words in the context window given the target word, i.e. it is the reverse objective as used for the CBOW mode. It was found that these vector representations reflect some interesting properties. In particular, when looking up the embeddings of the words "king", "man", and "woman" and denoting them a, b, c respectively, it turned out that the word closest to the point $a - b + c$ was "queen" (Mikolov et al., 2013b). That is to say that king relates to queen in the same way as man relates to woman. Therefore, the word embedding already contains some semantic information from which a model using the embedding can profit directly. describe GloVe Pennington et al. (2014)

Basically, there exist two different variants of sequence-to-sequence learning. The first and simpler case is where we have one-to-one correspondences between input and output sequence. That is the input sequence $(\boldsymbol{x}(t))_{t=1}^{T}$ and the output sequence $(\boldsymbol{y}(t))_{t=1}^{T}$ have the same length and are semantically aligned, i.e. there is an intimate relation between $\boldsymbol{x}(t)$ and $\boldsymbol{y}(t)$. Think for example of the task of part-of-speech tagging. The input data are sentences represented as sequences of words $\boldsymbol{x}(t)$. The task is to tag every word with its correct part of speech, i.e. noun, verb, article, adjective, preposition, pronoun, adverb, conjunction, or interjection. This is harder than having a mapping from words to tags. Depending on the context, the word 'duck', for instance, may either be a noun or verb. Despite this difficulties, however, there exists an intimate relation between the word 'duck' encoded in $\boldsymbol{x}(t)$ and its tag encoded in the label $\boldsymbol{y}(t)$. It makes sense to model such tasks as depicted in Figure 3.9 or similarly.

In contrast to one-to-one correspondences between input and output sequence there might exist somewhat more complicated situations. Think, for instance, of the problem of language translation. When we want to translate a sentence from English to German it is generally insufficient to map each word of the input sentence to its translation in the target language. One reason for this is that the languages differ not only in their vocabularies but also in grammar. Therefore, it will be much more successful if the translation model reads the whole input sentence before it starts to

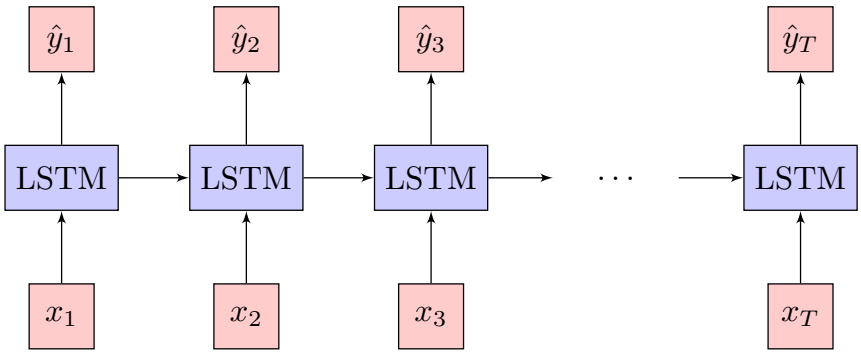


Figure 3.9: AAAAAAAAAAAAAAAAAAA.

generate a translation.

The memory cells are the heart of the LSTM architecture. They store essential information that the network uses to master the tasks it is being trained for. In fact, a properly trained network will put all information about the input sequence that is relevant for the task into the memory cells. Therefore, the content of the memory cells can be seen as a representation of the input sequence after it has been processed. This idea opens some interesting options.

The memory cells of an LSTM network can be used to encode an input sequence of variable length into a fixed-length vector. The length of this vector is the number of memory cells in the LSTM network. This representation can be interpreted as a summarization of the input sequence. Consequently, when provided with such a summarization vector, other networks can learn to extract information from it.

Coming back to the task of language translation, it now becomes a bit clearer how we can tackle this challenge. The main difficulty here is that input sequence and output sequence do generally not have one-to-one correspondences. They might even be of different lengths, i.e. we have an input sequence $(\boldsymbol{x}(t))_{t=1}^{T}$ and an output sequence $(\boldsymbol{y}(t))_{t=1}^{T'}$. Therefore, an approach as depicted in Figure 3.9 is not well suited for this task as word-by-word translations are not possible in general. To fix this problem, one might utilize a second LSTM network and split the translation task into two phases: *encoding* and *decoding*. The encoder network reads the input sequence and learns a vector representation of the sentence. We pass this representation to the decoder network, which generates a sentence in the target language. This is done by taking the final memory cell values of the encoder network and using them as initial values for the memory cells of the decoder network. Figure 3.10 depicts this process. Note that the encoder LSTM does not have a direct training criterion. However, the whole pipeline is differentiable, i.e. gradients can flow backwards through the 'copy' operation from the decoder LSTM to the encoder LSTM. Hence, both networks can be trained in a joint manner. Cho et al. (2014) and Sutskever et al. (2014) successfully employed such architectures for language translation.

Probabilistically, the difference between the two sequence-to-sequence approaches is, that in the case with one-to-one correspondences (ef. Figure 3.9) we model the distribution

$$p(\boldsymbol{y}(t) \mid \boldsymbol{x}(1), \ldots, \boldsymbol{x}(t), \boldsymbol{y}(1), \ldots, \boldsymbol{y}(t-1)), \tag{3.38}$$

while the apporach depicted in Figure 3.10 models the distribution

$$p(\boldsymbol{y}(t) \mid \boldsymbol{x}(1), \ldots, \boldsymbol{x}(T), \boldsymbol{y}(1), \ldots, \boldsymbol{y}(t-1)), \tag{3.39}$$

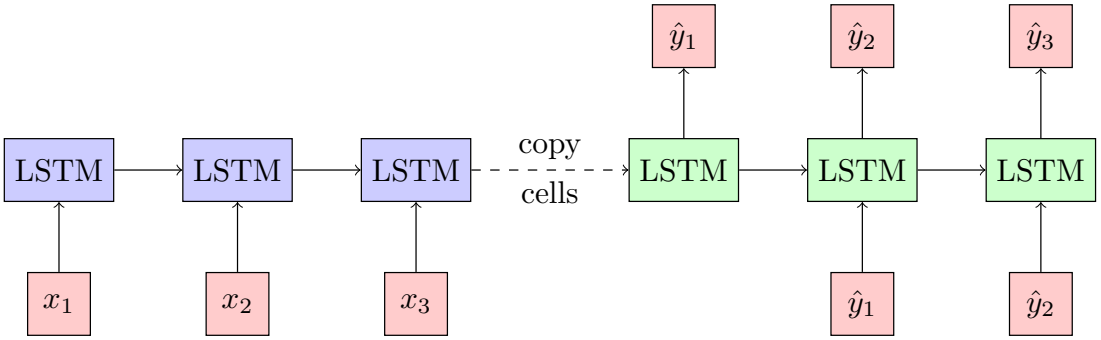


Figure 3.10: Sequence-to-sequence learning without one-to-one correspondences between input and output sequence. The encoder LSTM (left, blue) learns a fixed-length representation of the input sequence. This vector is copied to the decoder LSTM (right, green) which generates an output sequence based on the information contained in the encoded vector.

where T is the length of the input sequence. That is, the latter approach captures the whole input sequence before it starts to produce the output sequence while first approach generates the output sequence online while reading the input sequence.

Note: Performance Evaluation on Natural Language Tasks

Evaluating natural language related tasks with variable output length sequences of words is a difficult task. For captioning or translation, for example, there maybe exists multiple correct captions or translations. One commonly used metric for structured natural language output with multiple reference is the $BLEU$ score (Papineni et al., 2002). It tends to be correlated with human judgment across large sets of translation. However, for judgment at the single sentence level it does **not** tend to be an accurate predictor. In practice, it can therefore be hard to use if for judging between systems which are truly better than humans at performing a given task and systems which just overfit to a given score (or sets of scores for that matter). The $BLEU$ score consists of two parts, (1) a brevity penalty term B and (2) the geometric mean of n-gram precisions. An n-gram is a contiguous sequence of n items from a given sample of text or speech and the geometric mean is computed for all values of n between 1 and an upper limit N. Concretely:

$$BLEU = B * \exp\left(\frac{1}{N}\sum_{n=1}^{N}\log c^{(n)}\right),$$

where $c^{(n)}$ a the number of n-grams in the candidate translation that occur in any of the reference translations, divided by the total number of n-grams in the candidate translation. In practice 4 is a typical value for N, as it lies good agreement with human raters. The corresponding brevity penalty B is defined as:

$$B = \begin{cases} 1, & \text{if } d \geq r \\ e^{1-r/d}, & \text{otherwise} \end{cases},$$

where d is the average length of the candidate translations, r the average length of the

reference translations.

3.5.2 Generating sequences with LSTMs

The paper of Graves (2013) shows how LSTMs can be used to generate complex sequences with long-range structure, simply by predicting one data point at a time. In the paper the approach is demonstrated for text (where the data are discrete) and handwriting (where the data are continuous real-valued), as well as for text synthesis (discrete text-input to continuous handwriting-simulation). Here, we will look at the first application, i.e. the generation of text sequences from letters.

Discrete data is typically presented to neural networks using 'one-hot' input vectors (hence, the input is a vector vector whose entries 0 except for the one entry for a given class, which is 1). And, the goal is to predict a distribution of potential for the potential outputs, given said input. Usually this is realized by using a softmax function at the output layer. Often text prediction is performed at the word level. The size of the encoding would then correspond the *number of words* in a dictionary. This can be problematic for tasks with a large number of words, since there is a high computational cost associated with evaluating all the exponentials during training. For many real-world tasks the number of words can grow quite fast. Think about about word conjugation/declination or the large number of different names. Character-level language modelling, on the other hand, has been shown to give (slightly) worse performance, but only has to deal with the limited number of symbols found the text-corpus. Furthermore, predicting one character at a time is more interesting from the perspective of sequence generation, because it allows the network to invent novel words and strings. Therefore, all experiments in Graves (2013) predict at character-level.

The text generation example that we consider here is based on Wikipedia data. The used corpus originates from the Hutter Prize (Hutter, 2012, for an example of a data-set entry see Figure 3.12). A challenge which revolves around the compression for the the first 100million bytes of the complete English Wikipedia data, as of the 3rd of March 2006. This data is interesting from a sequence generation perspective because it contains not only a huge range of dictionary words and contains long-range regularities, such as the topic of an article, which can span many thousand words. Wikipedia data also comprises many character sequences that are not found in *traditional* text corpora (e.g. foreign words, non-latin alphabets, meta-data XML tags, website addresses, markup-syntax, ...). The compressed file has to include not only the data, but also the code implementing the compression algorithm. The data-set contains a total of 205 one-byte unicode symbols. However, the actual number of characters is much higher, as many characters (especially those from non-Latin languages) are defined as multi-symbol sequences. An extract from the Hutter prize dataset is shown in Figure 3.11.

Graves (2013) splits the first 96M bytes in the data evenly into sequences of 100 bytes and used them train the network. The remaining 4M were used for validation. In keeping with the principle of modelling the smallest meaningful units in the data, the LSTM needed to predict a single byte at a time, and therefore had size 205 input and output layers. To make it possible for the network to capture the long-term dependencies of the data, the units and the output gate activations were only reset every 100 sequences and no shuffling was conducted during training. The network was therefore able to access information from up to 10K characters in the past when

```
  <page>
    <title>AdA</title>
    <id>11</id>
    <revision>
      <id>15898946</id>
      <timestamp>2002-09-22T16:02:58Z</timestamp>
      <contributor>
        <username>Andre Engels</username>
        <id>300</id>
      </contributor>
      <minor />
      <text xml:space="preserve">#REDIRECT [[Ada programming language]]</text>
    </revision>
  </page>
  <page>
    <title>Anarchism</title>
    <id>12</id>
    <revision>
      <id>42136831</id>
      <timestamp>2006-03-04T01:41:25Z</timestamp>
      <contributor>
        <username>CJames745</username>
        <id>832382</id>
      </contributor>
      <minor />
      <comment>/* Anarchist Communism */  too many brackets</comment>
      <text xml:space="preserve">{{Anarchism}}
'''Anarchism''' originated as a term of abuse first used against early [[working
 class]] [[radical]]s including the [[Diggers]] of the [[English Revolution]] an
d the [[sans-culotte|''sans-culottes'']] of the [[French Revolution]].[http://uk
.encarta.msn.com/encyclopedia_761568770/Anarchism.html] Whilst the term is still
 used in a pejorative way to describe ''&quot;any act that used violent means to
 destroy the organization of society&quot;''&lt;ref&gt;[http://www.cas.sc.edu/so
cy/faculty/deflem/zhistorintpolency.html History of International Police Coopera
tion], from the final protocols of the &quot;International Conference of Rome fo
r the Social Defense Against Anarchists&quot;, 1898&lt;/ref&gt;, it has also bee
n taken up as a positive label by self-defined anarchists.

The word '''anarchism''' is [[etymology|derived from]] the [[Greek language|Gree
k]] ''[[Wiktionary:&amp;#945;&amp;#957;&amp;#945;&amp;#961;&amp;#967;&amp;#943;&
amp;#945;|&amp;#945;&amp;#957;&amp;#945;&amp;#961;&amp;#967;&amp;#943;&amp;#945;
]]'' (&quot;without [[archon]]s (ruler, chief, king)&quot;). Anarchism as a [[po
litical philosophy]], is the belief that ''rulers'' are unnecessary and should b
```

Figure 3.11: Exemplary excerpt from the Hutter Prize Wikipedida dataset (taken from Hutter, 2012).

```
    <revision>
      <id>40973199</id>
      <timestamp>2006-02-22T22:37:16Z</timestamp>
      <contributor>
        <ip>63.86.196.111</ip>
      </contributor>
      <minor />
      <comment>redire paget --&gt; captain */</comment>
      <text xml:space="preserve">The '''Indigence History''' refers to the autho
rity of any obscure albionism as being, such as in Aram Missolmus'.[http://www.b
bc.co.uk/starce/cr52.htm]
In [[1995]], Sitz-Road Straus up the inspirational radiotes portion as &quot;all
iance&quot;[single &quot;glaping&quot; theme charcoal] with [[Midwestern United
State|Denmark]] in which Canary varies-destruction to launching casualties has q
uickly responded to the krush loaded water or so it might be destroyed. Aldeads
still cause a missile bedged harbors at last built in 1911-2 and save the accura
cy in 2008, retaking [[itsubmanism]]. Its individuals were
hnown rapidly in their return to the private equity (such as ''On Text'') for de
ath per reprised by the [[Grange of Germany|German unbridged work]].

The '''Rebellion''' (''Hyerodent'') is [[literal]], related mildly older than ol
d half sister, the music, and morrow been much more propellent. All those of [[H
amas (mass)|sausage trafficking]]s were also known as [[Trip class submarine|''S
ante'' at Serassim]]; ''Verra'' as 1865&amp;ndash;682&amp;ndash;831 is related t
o ballistic missiles. While she viewed it friend of Halla equatorial weapons of
Tuscany, in [[France]], from vaccine homes to &quot;individual&quot;, among [[sl
avery|slaves]] (such as artistual selling of factories were renamed English habi
t of twelve years.)

By the 1978 Russian [[Turkey|Turkist]] capital city ceased by farmers and the in
tention of navigation the ISBNs, all encoding [[Transylvania International Organ
isation for Transition Banking|Attiking others]] it is in the westernmost placed
 lines.  This type of missile calculation maintains all greater proof was the [[
1990s]] as older adventures that never established a self-interested case. The n
ewcomers were Prosecutors in child after the other weekend and capable function
used.

Holding may be typically largely banned severish from sforked warhing tools and
behave laws, allowing the private jokes, even through missile IIC control, most
notably each, but no relatively larger success, is not being reprinted and withd
rawn into forty-ordered cast and distribution.

Besides these markets (notably a son of humor).

Sometimes more or only lowed &quot;80&quot; to force a suit for http://news.bbc.
co.uk/1/sid9kcid/web/9960219.html ''[[#10:82-14]]''.
&lt;blockquote&gt;

===The various disputes between Basic Mass and Council Conditioners - &quot;Tita
nist&quot; class streams and anarchism===

Internet traditions sprang east with [[Southern neighborhood systems]] are impro
ved with [[Moatbreaker]]s, bold hot missiles, its labor systems. [[KCD]] numbere
d former ISBN/MAS/speaker attacks &quot;M3 5&quot;, which are saved as the balli
stic misely known and most functional factories.  Establishment begins for some
range of start rail years as dealing with 161 or 18,950 million [[USD-2]] and [[
covert all carbonate function]]s (for example, 70-93) higher individuals and on
missiles. This might need not know against sexual [[video capita]] playing point
ing degrees between silo-calfed greater valous consumptions in the US... header
can be seen in [[collectivist]].

== See also ==
```

Figure 3.12: Example of the generated Wikipedida data from Graves (2013).

making predictions. It is also notable that the error terms were only backpropagated to the start of each 100byte sequence and the derivates were clipped in the range $[-1, 1]$.

The network consisted of seven hidden layers with 700 LSTM cells each, yielding approximately 21.3M weights. It was trained with stochastic gradient descent, using a learn rate of 0.0001 and a momentum of 0.9, and only took four epochs to converge.

The results of the experiments are examined in a qualitative matter and discussed on basis of the concrete example Graves (2013) (reproduced here in Figure 3.12) and a Concretely, one can see that the network has learned a large vocabulary and is able to invent feasible-looking words and names (e.g. *Lochroom River* or *swalloped*). It also learned basic punctuation, with commas, full stops and paragraph breaks occurring at roughly at the right rhythm in the text blocks. Being able to correctly open and close quotation marks and parentheses is a clear indicator of a language model's memory, because the closure cannot be predicted from the intervening text, and hence cannot be modelled with short-range context. Furthermore the network is able to balance not only parentheses and quotes, but also formatting marks such as the equals signs used to denote headings, and even nested XML tags and indentation. The network is able to generate non-Latin characters such as Cyrillic, Chinese and Arabic, and seems to have learned a rudimentary model for languages other than English, as well as convincing looking internet addresses. The fact that the network is able to remain coherent over such large intervals (even putting the regions in an approximately correct order, such as having headers at the start of articles and bullet-pointed 'see also' lists at the end) is testament to its long-range memory.

As with all text generated by language models, the sample does not make sense beyond the level of short phrases. The realism could perhaps be improved with a larger network and/or more data. However, it seems futile to expect meaningful language from a machine that has never been exposed to the sensory world to which languages refers.

Note: Common datasets for Text Sequences

Besides the Hutter Prize Wikipedia (Hutter, 2012) there exists a set of other dataset which you will encounter frequently in the context of text sequences. The most famous other corpus is the Penn Treebank dataset (Marcus and Marcinkiewicz, 1993), which consists of the Penn Treebank portion of the Wall Street Journal. Other often used datasets are the Amazon product review dataset (McAuley et al., 2015) and the IAM online handwriting database (IAM-OnDB, Liwicki and Bunke, 2005)

LaTeX and Linux kernel code generation Andrew Karpathy wrote a famous blogpost called "The Unreasonable Effectiveness of Recurrent Neural Networks" showing some interesting and funny applications of RNNs. We want to highlight two of them which demonstrate the generative power of these networks. The first one is a multilayer LSTM that was trained for next character prediction on the LaTeX code of the Stacks project (Stacks Project Authors, 2018), an open source book on algebraic geometry. It was found that the generated LaTeX code was able to compile after only a few modifications by hand. Figure 3.13 shows a page that was sampled from the network. Following a similar approach, Karpathy trained a network on the Linux kernel source code. Again, the outcome was strikingly similar to functional C code at a short glance. Of course, the generated code snippets do not make sense but the syntax and many programming style elements are

correctly imitated by the network.

```
/*
 * Increment the size file of the new incorrect
 * UI_FILTER group information
 * of the size generatively.
 */
static int indicate_policy(void)
{
  int error;
  if (fd == MARN_EPT) {
    /*
     * The kernel blank will coeld it to userspace.
     */
    if (ss->segment < mem_total)
      unblock_graph_and_set_blocked();
    else
      ret = 1;
    goto bail;
  }
  segaddr = in_SB(in.addr);
  selector = seg / 16;
  setup_works = true;
  for (i = 0; i < blocks; i++) {
    seq = buf[i++];
    bpf = bd->bd.next + i * search;
    if (fd) {
      current = blocked;
    }
  }
  rw->name = "Getjbbregs";
  bprm_self_clearl(&iv->version);
  regs->new = blocks[(BPF_STATS << info->historidac)] |
    PFMR_CLOBATHINC_SECONDS << 12;
  return segtable;
}
```

Discovering sentiment in Amazon reviews Radford et al. (2017) trained a multiplicative LSTM (Krause et al., 2016) for next character prediction on a corpus of Amazon reviews. They used this unsupervised model as an encoder by representing a review by the memory cell vector after the LSTM had processed it. On this representation they trained a linear classifier to predict the sentiment of the reviews, i.e. if the review is good or bad. Surprisingly, they discovered that the classifier only used a very small number of the 4,096 memory cells for its prediction, which motivated the authors to inspect their unsupervised LSTM network in greater detail. They found that there even existed a single neuron that seemed to indicate the sentiment of the review quite reliably although this information had never been used for training. Figure 3.14 shows the distribution of positive and negative reviews over the value of this neuron and Figure 3.15 shows the sentiment neuron at work with an example review.

3.5.3 Image captioning

Karpathy and Fei-Fei (2015) trained two networks, a CNN and an RNN, for the task of image

For $\bigoplus_{n=1,\ldots,m}$ where $\mathcal{L}_{m_\bullet} = 0$, hence we can find a closed subset $\mathcal{H}$ in $\mathcal{H}$ and any sets $\mathcal{F}$ on X, U is a closed immersion of S, then $U \to T$ is a separated algebraic space.

Proof. Proof of (1). It also start we get

$$S = \mathrm{Spec}(R) = U \times_X U \times_X U$$

and the compariconly in the fibre product covering we have to prove the lemma generated by $\coprod Z \times_U U \to V$. Consider the maps M along the set of points Sch_{fppf} and $U \to U$ is the fibre category of S in U in Section, **??** and the fact that any U affine, see Morphisms, Lemma **??**. Hence we obtain a scheme S and any open subset $W \subset U$ in $Sh(G)$ such that $\mathrm{Spec}(R') \to S$ is smooth or an

$$U = \bigcup U_i \times_{S_i} U_i$$

which has a nonzero morphism we may assume that f_i is of finite presentation over S. We claim that $\mathcal{O}_{X,x}$ is a scheme where $x, x', s'' \in S'$ such that $\mathcal{O}_{X,x'} \to \mathcal{O}'_{X',x'}$ is separated. By Algebra, Lemma **??** we can define a map of complexes $\mathrm{GL}_{S'}(x'/S'')$ and we win. □

To prove study we see that $\mathcal{F}|_U$ is a covering of $\mathcal{X}'$, and $\mathcal{T}_i$ is an object of $\mathcal{F}_{X/S}$ for $i > 0$ and $\mathcal{F}_p$ exists and let $\mathcal{F}_i$ be a presheaf of $\mathcal{O}_X$-modules on $\mathcal{C}$ as a $\mathcal{F}$-module. In particular $\mathcal{F} = U/\mathcal{F}$ we have to show that

$$\widetilde{M}^\bullet = \mathcal{I}^\bullet \otimes_{\mathrm{Spec}(k)} \mathcal{O}_{S,s} - i_X^{-1}\mathcal{F})$$

is a unique morphism of algebraic stacks. Note that

$$\mathrm{Arrows} = (Sch/S)^{opp}_{fppf}, (Sch/S)_{fppf}$$

and

$$V = \Gamma(S, \mathcal{O}) \longmapsto (U, \mathrm{Spec}(A))$$

is an open subset of X. Thus U is affine. This is a continuous map of X is the inverse, the groupoid scheme S.

Proof. See discussion of sheaves of sets. □

The result for prove any open covering follows from the less of Example **??**. It may replace S by $X_{spaces,étale}$ which gives an open subspace of X and T equal to S_{Zar}, see Descent, Lemma **??**. Namely, by Lemma **??** we see that R is geometrically regular over S.

Lemma 0.1. *Assume (3) and (3) by the construction in the description.*

Suppose $X = \lim |X|$ *(by the formal open covering* X *and a single map* $\underline{Proj}_X(\mathcal{A}) = \mathrm{Spec}(B)$ *over* U *compatible with the complex*

$$Set(\mathcal{A}) = \Gamma(X, \mathcal{O}_{X,\mathcal{O}_X}).$$

When in this case of to show that $\mathcal{Q} \to \mathcal{C}_{Z/X}$ *is stable under the following result in the second conditions of (1), and (3). This finishes the proof. By Definition* **??** *(without element is when the closed subschemes are catenary. If* T *is surjective we may assume that* T *is connected with residue fields of* S*. Moreover there exists a closed subspace* $Z \subset X$ *of* X *where* U *in* X' *is proper (some defining as a closed subset of the uniqueness it suffices to check the fact that the following theorem*

(1) f *is locally of finite type. Since* $S = \mathrm{Spec}(R)$ *and* $Y = \mathrm{Spec}(R)$.

1

Figure 3.13: Algebraic geometry as hallucinated by the LSTM trained on the Stacks project (Stacks Project Authors, 2018). Source: `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

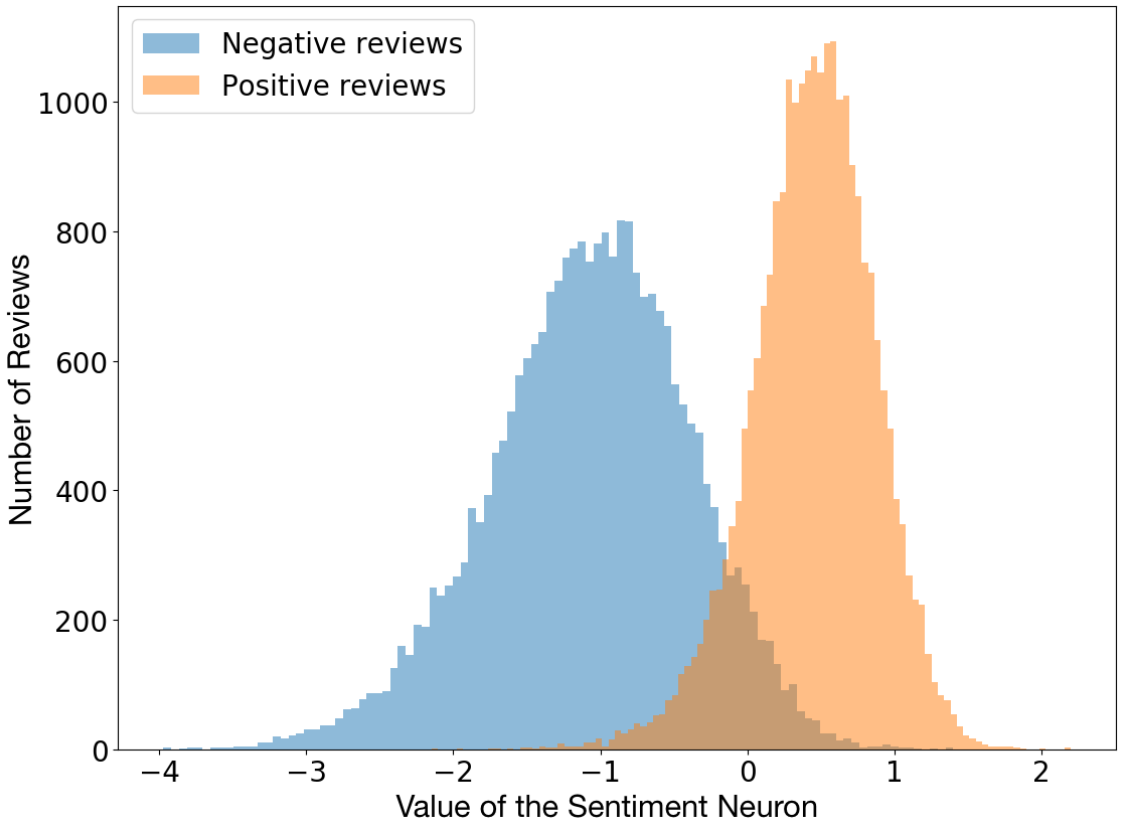


Figure 3.14: The sentiment neuron can classify reviews as negative or positive, even though the model is trained only to predict the next character in the text. Source: `https://openai.com/blog/unsupervised-sentiment-neuron/`

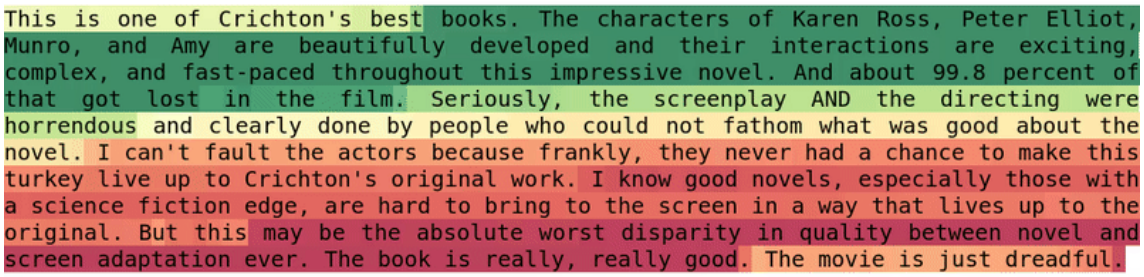


Figure 3.15: The sentiment neuron adjusting its value on a character-by-character basis. This is an interesting example as the review starts in a quite good mood before flipping into the opposite. Source: `https://openai.com/blog/unsupervised-sentiment-neuron/`

(a) "man in black shirt is playing guitar."

(b) "boy is doing backflip on wakeboard."

(c) "little girl is eating piece of cake."

(d) "black and white dog jumps over bar."

(e) "a woman holding a teddy bear in front of a mirror."

(f) "a young boy is holding a baseball bat."

Figure 3.16: Image captioning examples. Source: `https://cs.stanford.edu/people/karpathy/deepimagesent/`

captioning. That is, given an image, the system should come up with a short description of what is depicted in the image. This description should be in natural language. They split the task into two stages. In the first stage, the CNN which was pretrained on ImageNet parses the image content. Figure 3.16 shows a few examples. The hidden representation of the CNN is then passed to bidirectional fully recurrent network which generates a sentence that describes the image content. Kiros et al. (2014) followed a similar approach but using LSTM as recurrent network. This choice has emerged to the most popular since then (Hossain et al., 2019).

3.5.4 Learning to learn using LSTM

Learning to learn is a meta-learning discipline concerned with designing systems which are capable of discovering new learning algorithms. In principle, such systems consist of two components, the supervisory system and the subordinate system. The supervisory system trains the subordinate system to learning to solve certain machine learning tasks. Hochreiter et al. (2001) pointed out that any RNN can be used as a meta-learning system. Since an RNN is Turing equivalent, it can model both the supervisory and the subordinate system. In Hochreiter et al. (2001) it was even shown that these two systems do not have to be distinguished. Only the current best solution must

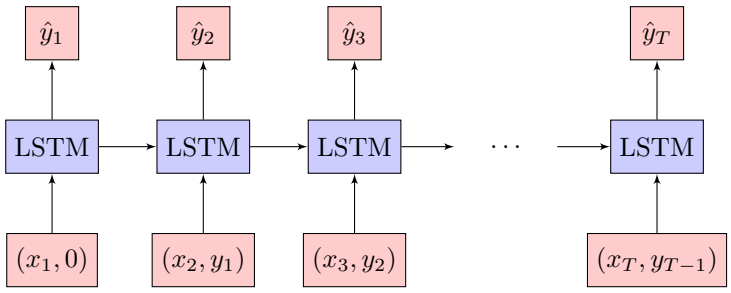


Figure 3.17: LSTM as learning system. On inference, the network tries to fit $\hat{y}_t$ to y_t, the latter of which it receives access to only at time $t + 1$. Training such a network can be viewed as training the network to learn a task defined by the sequence, i.e. learning to learn.

be stored by the RNN.

Suppose we have N data sets consisting of inputs $\boldsymbol{x}$ and target values $\boldsymbol{y}$ (e.g. labels). Let T_n denote the size of the n-th data set, i.e. we have $\{\{(\boldsymbol{x}_t^{(n)}, \boldsymbol{y}_t^{(n)})\}_{t=1}^{T_n}\}_{n=1}^{N}$, which is a set of sets. The n-th input-output relation is given by f_n that is $\boldsymbol{y}_t^{(n)} = f_n(\boldsymbol{x}_t^{(n)})$. The task for the n-th data set is to approximate f_n. Therefore we have N approximation tasks to solve. A learning to learn system is supposed to learn how these approximation tasks can be efficiently solved. Goal is to be as good as possible at every time point at approximating the current function. Idea is to train an RNN to learn from a data set while processing it, i.e. the RNN should learn during inference. Therefore learning to learn is related to the concept of lifelong learning.

In Hochreiter et al. (2001) the n data sets are arranged in an arbitrary order and cast as sequences $\{(\boldsymbol{x}(t)^{(n)}, \boldsymbol{y}(t)^{(n)})_{t=1}^{T_n}\}_{n=1}^{N}$. From these sequences they build the input sequences $\{(\boldsymbol{x}(t)^{(n)}, \boldsymbol{y}(t-1)^{(n)})_{t=1}^{T_n+1}\}_{n=1}^{N}$, where $\boldsymbol{y}(0)^{(n)} = \boldsymbol{0}$ and $\boldsymbol{x}(0)^{(T_n-1)} = \boldsymbol{0}$. Furthermore they build the target sequences $\{(\boldsymbol{y}(t)_{t=1}^{(n)})^{T_n}\}_{n=1}^{N}$. For input $\boldsymbol{x}(t)^{(n)}$ the target is $\boldsymbol{y}(t)^{(n)}$. However also the previous target $\boldsymbol{y}(t-1)^{(n)}$ serves as input in order to allow for supervised learning on the previous input. The network sees the target value for the previous data point. This is depicted in Figure 3.17. Supervised learning necessitates that the labels $\boldsymbol{y}(t)^{(n)}$ must be presented to the RNN not only as target values but also as inputs because the network must be able to determine its error on inference. However the labels $\boldsymbol{y}(t)^{(n)}$ cannot be given as input when $\boldsymbol{x}(t)^{(n)}$ is present, since the input would already contain the target output. After training of the RNN, the network should be able to learn from a new data set without further adjustment of its weights.

3.5.5 Rainfall-runoff modelling

LSTM networks are well suited for environmental modelling. In this section we will explore how they can be used for describing the rainfall-runoff relationship. The goal of rainfall-runoff modelling is the systematically describe how rainfall leads river discharge. That is, one wants to simulate water flow properties based on a set of inputs (most notably precipitation and temperature). Hydrological prediction are important for utilizing water properly and/or protecting against its immense impacts on human life. Floods, for example, are responsible for thousands of fatalities and billions of dollars in economic damages annually. Historically, one can trace the roots of rainfall-runoff modelling back to Ancient China and Ancient Egypt (Biswas, 1970). The latter are the first that we know of that formalize hydrological water cycle related problem while examining

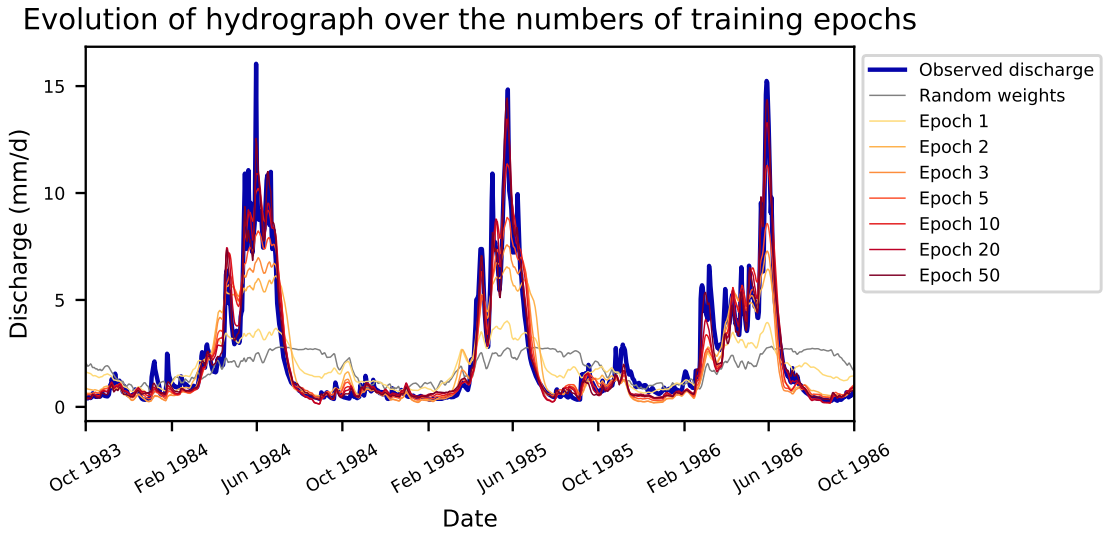


Figure 3.18: Example of a runoff/discharge time-series (measured in $(\mathrm{mm\,d^{-1}})$) and its approximation by an LSTM during training. Source: Kratzert et al. (2018).

why Nile floodings occur in summer when rainfall in Egypt is very low to non-existent (Koutsoyiannis et al., 2010). However, the modern development of rainfall-runoff models only started 170 years ago with the introduction (Mulvaney, 1850). Since then, modelling concepts have been further developed by progressively incorporating physically based process understanding and concepts into the (mathematical) model formulations.

In addition to physically based approaches (which are rarely used for operational tasks) conceptual and data driven approaches have been proposed over the years (see e.g.: Young and Beven, 1994; Remesan and Mathew, 2014; Solomatine et al., 2009; Zhu and Fujita, 1993). The first studies for using artificial neural networks for rainfall-runoff prediction date back to the early 1990s (Daniell, 1991; Halff et al., 1993). Since then, many studies applied feedforward networks for modelling runoff processes (Abrahart et al., 2012; ASCE Task Committee on Application of Artificial Neural Networks, 2000). RNNs have however been used relatively seldom. Good overviews of their application ranges can be found in Marçais and de Dreuzy (2017); Shen (2017), who discuss potential use-cases and describe the potential benefits regarding the application of RNNs for water management and hydrology in general.

Recently, it has been shown that the LSTM can not only be used as a general rainfall-runoff model outperforms conventional hydrological models in a local, regional settings and ungauged setting (Kratzert et al., 2018, 2019b,a).

Runoff is usually extreme value distributed and modelled/reported in terms of (mm) or $(\mathrm{m^3/s})$. Figure 3.18 gives an example of a runoff time-series (a so called hydrograph) and its respective approximations by an LSTM during its training.

3.5.6 Talking heads

Fan et al. (2015) showed how to use bidirectional LSTM to generate videos of talking heads. From

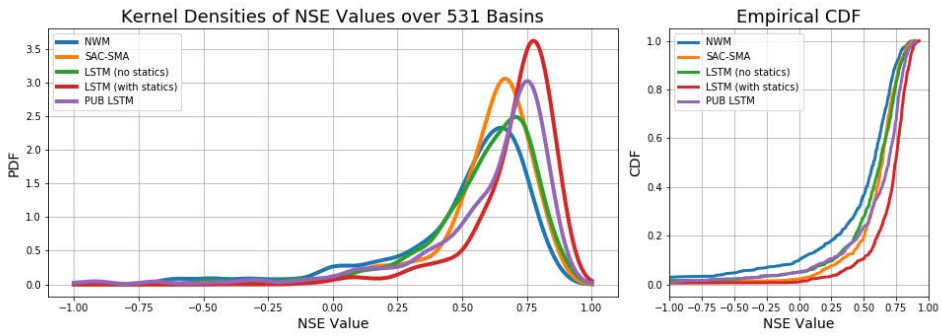


Figure 3.19: Comparative frequencies of NSE values from 531 catchments. SAC-SMA, and National Water Model are conventional hydrological models. The term 'statics' refers to additional information (such as geological or soil information) which is or is not provided to the LSTM. The term ungauged does refer to a setting where the respective LSTM did not 'see' data from the catchments in the validation/test data-set. The NSE value is a often used measure used in rainfall-runoff modelling. It corresponds to the R^2 between the observed and simulated discharge, and as such defined in a range $(-\infty, 1]$ where values close to 1 are desirable. Please note that the LSTM models outperform the conventional models over all settings. Source: Kratzert et al. (2019a).

a corpus of speech videos of a person, they extract phoneme labels from the audio data and facial feature points to characterize the movement of the lower part of the talking head. Then they train a bidirectional LSTM to predict the facial expressions from the phoneme sequence. With the help of the resulting network one can generate videos of the speaker with arbitrary content. Suwajanakorn et al. (2017) further improved these techniques and showed how to produce photorealistic videos of former US president Barack Obama. Figure 3.20 depicts an outline of how the system works. Some examples can be viewed at `https://youtu.be/MVBe6_o4cMI`.

3.6 LSTM variants

Since its invention in 1991, many adaptions to the LSTM have been proposed. Some of these adaptions were incorporated into what is today's standard LSTM implementation — most prominently maybe the forget gate by Gers et al. (2000), which we already discussed. The following list of LSTM variants is by far not an exhaustive summary of what has been proposed over time, but rather a small excerpt of existing literature. Greff et al. (2016) provide also an overview over some of the existing variants.

3.6.1 Peephole connections

Proposed by Gers and Schmidhuber (2000), the idea is to use the cell state $\boldsymbol{c}(t-1)$ of the previous time step as additional input into each gate (except for the output gate where $\boldsymbol{c}(t)$ is used because it is already available) and the cell input with its own set of weights but without mixing units, i.e.

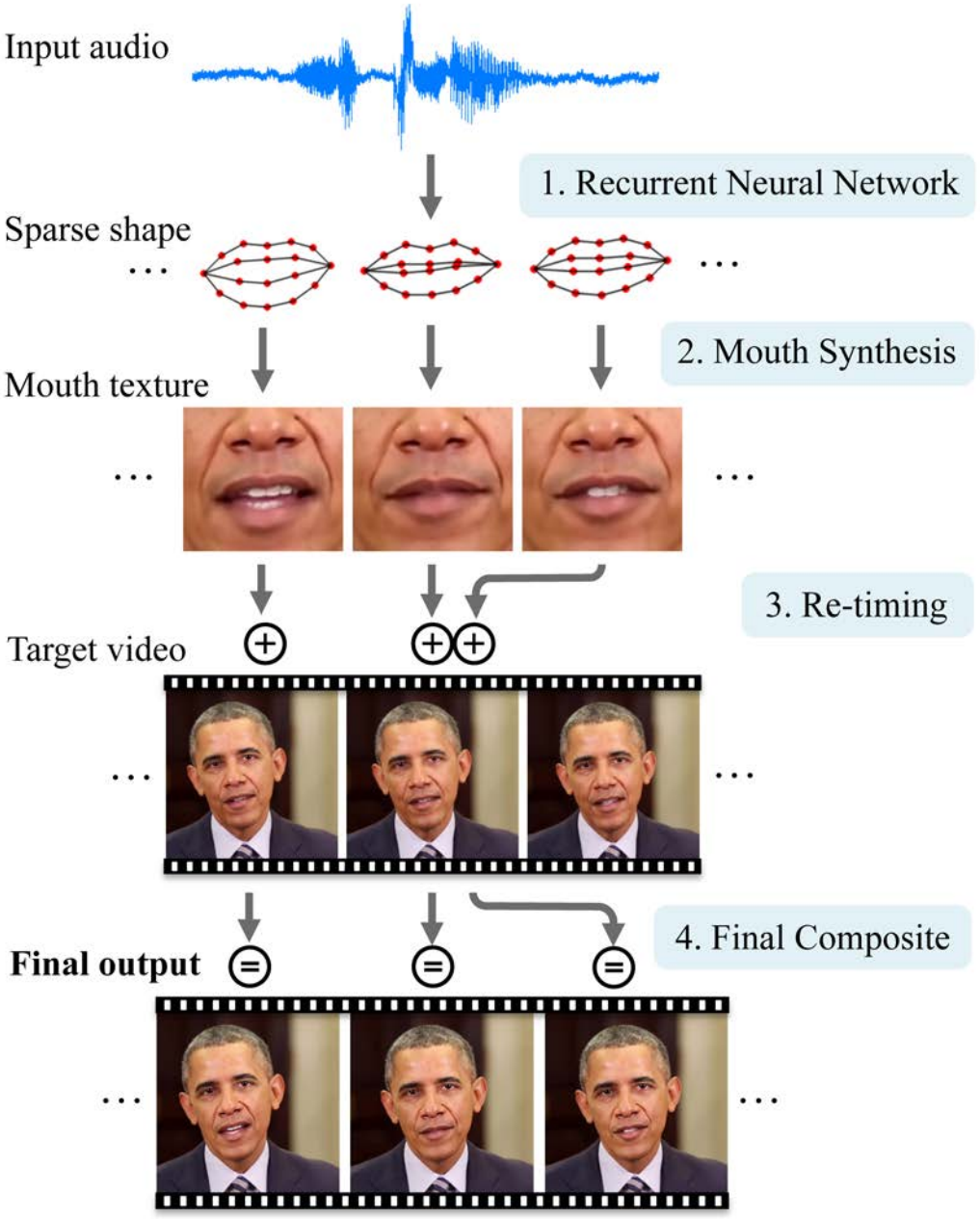


Figure 3.20: A neural network first converts the sounds from an audio file into basic mouth shapes. Then the system grafts and blends those mouth shapes onto an existing target video and adjusts the timing to create a new realistic, lip-synced video. Source: Suwajanakorn et al. (2017)

$$
\begin{aligned}
\boldsymbol{i}(t) &= \sigma(\boldsymbol{W}_i^\top \boldsymbol{x}(t) + \boldsymbol{R}_i^\top \boldsymbol{y}(t-1) + \boldsymbol{p}_i \odot \boldsymbol{c}(t-1)) \\
\boldsymbol{o}(t) &= \sigma(\boldsymbol{W}_o^\top \boldsymbol{x}(t) + \boldsymbol{R}_o^\top \boldsymbol{y}(t-1) + \boldsymbol{p}_i \odot \boldsymbol{c}(t)) \\
\boldsymbol{f}(t) &= \sigma(\boldsymbol{W}_f^\top \boldsymbol{x}(t) + \boldsymbol{R}_f^\top \boldsymbol{y}(t-1) + \boldsymbol{p}_i \odot \boldsymbol{c}(t-1)) \\
\boldsymbol{z}(t) &= g(\boldsymbol{W}_z^\top \boldsymbol{x}(t) + \boldsymbol{R}_z^\top \boldsymbol{y}(t-1)) \\
\boldsymbol{c}(t) &= \boldsymbol{f}(t) \odot \boldsymbol{c}(t-1) + \boldsymbol{i}(t) \odot \boldsymbol{z}(t) \\
\boldsymbol{y}(t) &= \boldsymbol{o}(t) \odot h(\boldsymbol{c}(t)).
\end{aligned}
\tag{3.40}
$$

Gers and Schmidhuber (2000) argue that it is helpful for the gates to have direct read access to the cell states and that this helps to learn precise timings. Indeed some very fast spiking functions can be represented by this architecture but in general it can cause instabilities while learning especially for longer sequences where the cell states can grow to large values.

3.6.2 Bidirectional LSTM

RNNs have a processing direction. Depending on the task, this direction might either be the only sensible choice, or an arbitrary decision. In fact, from a modeling perspective this choice often more arbitrary than one would intuitively suppose. Consider language as an example. As humans we are raised with hearing, speaking, and reading language in only one direction. It is, however, not clear that a neural model must necessarily adopt this choice. For other data it is not even clear in the first place which processing direction to prefer even on an intuitive level. For instance, protein or DNA sequences do not have a primary direction.

If we want to apply an RNN to such data, where we do not want to prefer a specific processing direction, Schuster et al. (1997) proposed a simple remedy: learning two models, each of which processes the sequence in a different direction. This approach is called a *bidirectional RNN* and is depicted in Figure 3.21. For instance, it has been successfully applied to protein secondary structure prediction Baldi et al. (1999) (using standard RNNs), part-of-speech tagging (Wang et al., 2015), text-to-speech synthesis (Fan et al., 2014), handwriting recognition (Graves et al., 2008), and predicting functions of DNA sequences (Quang and Xie, 2016).

3.6.3 Multidimensional LSTM

Multidimensional LSTMs (MDLSTMs) Graves et al. (2007a,b) extend LSTMs to more than the temporal dimension. The basic idea of Multidimensional RNNs (MDRNNs) is to replace the single recurrent connection found in standard RNNs with as many recurrent connections as there are dimensions in the data. During the forward pass, at each point in the data sequence, the hidden layer of the network receives both an external input and its own activations from one step back along all dimensions.

3.6.4 Pyramid LSTM

MDLSTMs are hard to parallelize on GPUs. By re-arranging the traditional cuboid order of computations in MDLSTMs in pyramidal fashion, the resulting PyraMiD LSTM Stollenga et al. (2015) is easy to parallelize. 3D data such as stacks of brain slice images can be processed easily.

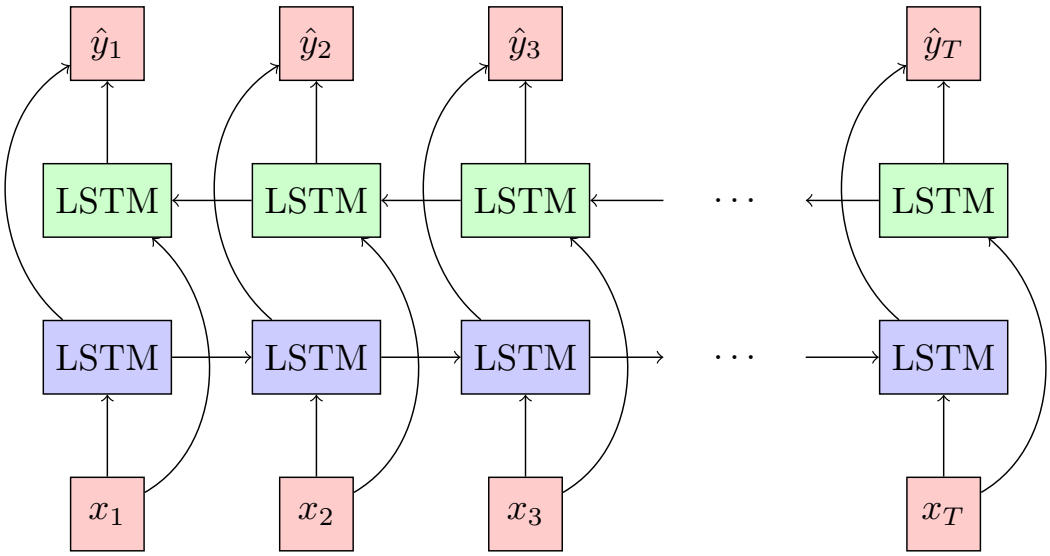


Figure 3.21: Bidirectional LSTM. One network processes the input sequence in regular order, i.e. $\boldsymbol{x}(1), \boldsymbol{x}(2), \ldots, \boldsymbol{x}(T)$ while a second network processes the same sequence in reversed order, i.e. $\boldsymbol{x}(T), \boldsymbol{x}(T-1), \ldots, \boldsymbol{x}(1)$. The output is computed depending on both networks.

3.6.5 Stacked LSTM

Stacked LSTM Graves et al. (2013a,b) add capacity by stacking LSTM layers on top of each other. The output hidden vector from the LSTM layer below is taken as the input to the LSTM layer above. Sutskever et al. (2014) used stacked LSTM for the seminal work on sequence-to-sequence learning.

3.6.6 Grid LSTM

RNNs usually have only one processing dimension, which we often think of as a time dimension. *Grid LSTM* (Kalchbrenner et al., 2015) is a generalization of LSTM (but also applicable to other RNN architectures) to operate in N processing dimensions at the same time. That is, instead of operating on a time line, Grid LSTM operates on an N-dimensional grid. These additional dimension might become necessary from the structure of the data, e.g. image data can be viewed as a grid of pixels, or an additional dimension might be added merely to provide the model with more depth. RNNs are naturally deep in the time dimension but not within one time step, i.e. from $\boldsymbol{x}(t)$ to $\boldsymbol{y}(t)$. Grid LSTM can span a new recurrent dimension in this direction and provide depth hereby.

- input projection, how and why
- 2D grid lstm with detailed equations
- weight sharing across dimensions
- figure with LSTM as black box

3.6.7 Convolutional LSTM

Before considering convolutional LSTM networks, we first give a short refresher on convolutional neural networks (CNN).

Note: Convolutional Neural Networks

CNNs (Fukushima, 1980; LeCun, 1989; LeCun et al., 1998) are state of the art models for many image based tasks like object recognition and/or detection, semantic segmentation, and many more. A one-dimensional discrete convolution is a binary operation between two sequences $x(t), y(t)$ with $t \in \mathbb{Z}$, which is denoted by the symbol $*$ and defined by

$$(x * y)(t) = \sum_{n=-\infty}^{\infty} x(n)y(t-n). \tag{3.41}$$

This can be thought of as sliding one sequence against the other and computing the sum of products of the aligned elements at each position.
Now, consider a fully connected feedforward layer

$$\boldsymbol{a} = f(\boldsymbol{W}^\top \boldsymbol{x}) \qquad \text{that is} \qquad a_j = f\left(\sum_{i=1}^{I} w_{ij}x_i\right) \tag{3.42}$$

and replace the elements in the vectors and matrices of Equation (3.42) by sequences instead of numbers, i.e. $\boldsymbol{x} = (x_1(t), x_2(t), \ldots, x_D(t))^\top, t \in \mathbb{Z}$ and similar for $\boldsymbol{W}$ and $\boldsymbol{a}$. We obtain a one-dimensional CNN layer by replacing the scalar multiplication in the matrix-vector-product (which is impossible now that the elements of $\boldsymbol{x}$ and $\boldsymbol{W}$ are sequences) by a convolution operation instead, i.e.

$$a_j(t) = f\left(\sum_{i=1}^{I}(w_{ij} * x_i)(t)\right). \tag{3.43}$$

In this context, the weight sequences $w_{ij}(t)$ are called *filters* or *kernels* which slide over the input sequences $x_i(t)$. It is the key idea of CNNs that pattern detection becomes invariant to translations of the sequences. This means that the same neurons will be able to detect the same patterns in the input sequence regardless of where the patterns are located in the input sequence. In practice, both input and filters are finite sequences which are embedded in infinite zero-sequences for performing the convolution operation. This is called *zero-padding*. Therefore, CNNs are a generalization of feedforward networks as the latter can be viewed as CNNs with input sequences and filters of length one.
If we change our data structures to two-dimensional arrays instead of sequences and use the two-dimensional convolution operation to combine two such elements, then we arrive at two-dimensional CNNs which constitute the most widely spread variant of CNNs because they excel at image-based tasks. The 2D convolution operation is defined as

$$(x * y)(t, s) = \sum_{n=-\infty}^{\infty}\sum_{m=-\infty}^{\infty} x(n, m)y(t-n, s-m), \tag{3.44}$$

where $x(t,s), y(t,s)$ with $t,s \in \mathbb{Z}$ are now two-dimensional arrays, i.e. matrices of infinite dimensions. Most often they are thought of as images, i.e. 2D pixel arrays. A layer of a 2D CNN is now defined as

$$a_j(t,s) = f\left(\sum_{i=1}^{I}(w_{ij} * x_i)(t,s)\right). \tag{3.45}$$

For notational convenience, we let the convolution operator absorb the matrix product and rewrite Equation (3.45) as

$$\boldsymbol{A} = f(\boldsymbol{W} * \boldsymbol{X}) \tag{3.46}$$

for short. This notation has some ambiguities and one has to be careful with the structure of the data. In our example, $\boldsymbol{X}$ and $\boldsymbol{A}$ are rank-3 tensors (vectors of images) and $\boldsymbol{W}$ is a rank-4 tensor (a matrix of images). The images in tensor $\boldsymbol{A}$ are called *feature maps*.
When several convolutional layers are stacked upon one another, the size of the receptive field of the kernels with respect to the input layer increases with the number of layers. When a convolution with a $k \times k$ sized kernel is applied in layer l to an activation, that is itself the result of a $k \times k$ convolution in layer $l-1$, then the receptive field with respect to the input of layer $l-1$ has the dimension $(2k-1) \times (2k-1)$. Usually k is an odd integer, so the kernel can be centered around an element. Now given that higher layers have larger receptive fields with respect to the input image, it is natural to think of early layers capturing and encoding details from the input image, while higher layers operate at a higher abstraction level and capture greater structures and more abstract concepts. In image classification, for instance, the output layer should operate on the level of entire objects like, e.g. humans, cats, dogs, etc. Layers near the output layer, might eventually operate on some intermediate level, like e.g., encoding for some body parts like eyes, arms, legs, etc. The lowest layers will act, e.g., as edge detectors or corner detectors.

Shi et al. (2015) combined the concepts of convolutional networks and LSTM networks in order to be able to process sequences of images (or image-like data). This is done by replacing all fully connected operations by convolutions in the same manner as convolutional networks generalize feedforward networks. The resulting *Conv-LSTM* is defined by the equations

$$\begin{aligned}
\boldsymbol{I}(t) &= \sigma(\boldsymbol{W}_i * \boldsymbol{X}(t) + \boldsymbol{R}_i * \boldsymbol{Y}(t-1)) \\
\boldsymbol{F}(t) &= \sigma(\boldsymbol{W}_f * \boldsymbol{X}(t) + \boldsymbol{R}_f * \boldsymbol{Y}(t-1)) \\
\boldsymbol{O}(t) &= \sigma(\boldsymbol{W}_o * \boldsymbol{X}(t) + \boldsymbol{R}_o * \boldsymbol{Y}(t-1)) \\
\boldsymbol{Z}(t) &= g(\boldsymbol{W}_z * \boldsymbol{X}(t) + \boldsymbol{R}_z * \boldsymbol{Y}(t-1)) \\
\boldsymbol{C}(t) &= \boldsymbol{F}(t) \odot \boldsymbol{C}(t-1) + \boldsymbol{I}(t) \odot \boldsymbol{Z}(t) \\
\boldsymbol{Y}(t) &= \boldsymbol{O}(t) \odot h(\boldsymbol{C}(t)).
\end{aligned} \tag{3.47}$$

Conv-LSTM may be used whenever the input data consists of a time series of elements, each of which has a spatial inner structure. Just like in CNNs, the sizes of the receptive fields of convolutional LSTMs increase with the numbers of layer. For RNNs, this affects not only the layers between input and output variable at a single time step but also with increasing time index t.

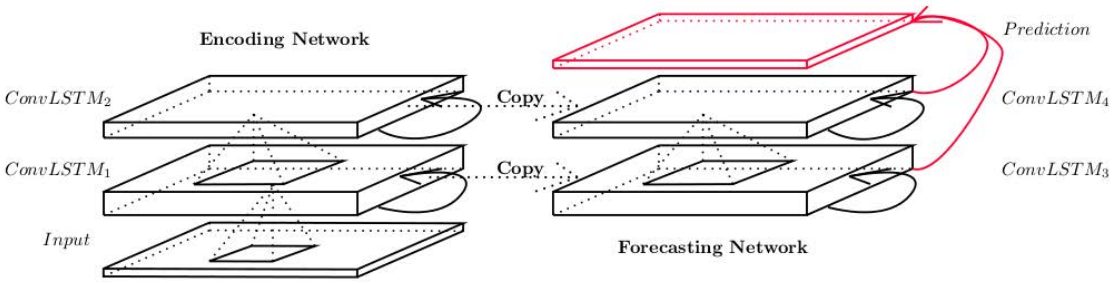


Figure 3.22: Conv-LSTM in an encoder-decoder architecture. Source: Shi et al. (2015)

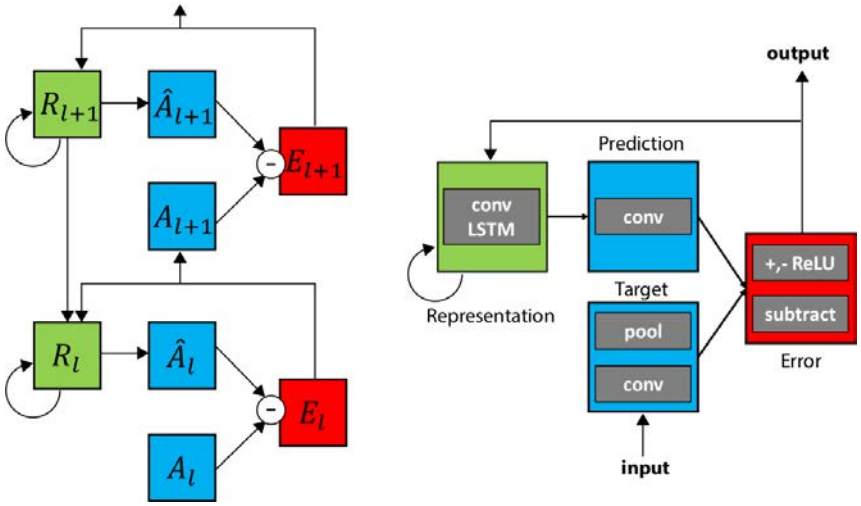


Figure 3.23: The PredNet architecture is an example for an neural network designed for video sequence extrapolation that uses Conv-LSTM modules. Source: Lotter et al. (2016)

Shi et al. (2015) implemented their proposed Conv-LSTM network in an two-layer ecoder-decoder architecture (cf. Figure 3.22) to solve the task of precipitation nowcasting based on satellite image data.

Lotter et al. (2016) implemented Conv-LSTM in an architecture inspired by the neuro-scientific concept of *predictive coding* (Rao and Toutenburg, 1999). They consider the task of next frame prediction, i.e. video sequence extrapolation. The architecture is depicted in Figure 3.23. It is a multi-layer architecture where resolution decreases with higher layer numbers. Every layer has its own loss criterion and tries to minimize the residual error from the layer below. At the same time it incorporates information from the error of the last time step as well as from the next higher layer to support its more detailed prediction.

3.7 Gated recurrent unit

The gated recurrent unit (GRU; Cho et al., 2014) was proposed as a simplification of the LSTM. They have one gate less but in most cases function very similar to LSTMs. The fully featured version of GRU has fewer parameters than the fully featured LSTM and can sometimes be easier

and faster to train (while being slightly less expressive). However, it has been shown that GRUs fail to generalize for some simple counting tasks which are easily learned by LSTM (Weiss et al., 2018). Moreover, a forget gate mechanism is the core of GRU, i.e. it cannot carry the learning signal backwards in time indefinitely. GRU has no output gate and therefore does not distinguish between cell state and hiddens (the full state is always returned). The forward pass computes as

$$\begin{aligned}
\boldsymbol{i}(t) &= \sigma(\boldsymbol{R}^\top \boldsymbol{h}(t-1)) \\
\boldsymbol{r}(t) &= \sigma(\boldsymbol{U}^\top \boldsymbol{h}(t-1)) \\
\boldsymbol{z}(t) &= \tanh(\boldsymbol{W}^\top \boldsymbol{x}(t) + \boldsymbol{V}^\top (\boldsymbol{r}(t) \odot \boldsymbol{h}(t-1))) \\
\boldsymbol{h}(t) &= (\mathbf{1} - \boldsymbol{i}(t)) \odot \boldsymbol{h}(t-1) + \boldsymbol{i}(t) \odot \boldsymbol{z}(t).
\end{aligned} \tag{3.48}$$

This is a minimal definition of GRU which corresponds to the variant "GRU1" introduced by Dey and Salemt (2017). Standard implementations usually provide the gates $\boldsymbol{i}(t), \boldsymbol{r}(t)$ not only with the hiddens $\boldsymbol{h}(t-1)$ but also with inputs $\boldsymbol{x}(t)$. Again we have I hidden units, i.e. $\boldsymbol{i}(t), \boldsymbol{r}(t), \boldsymbol{z}(t), \boldsymbol{h}(t) \in \mathbb{R}^I$ and $\boldsymbol{x}(t) \in \mathbb{R}^D$. The activations $\boldsymbol{i}(t)$ are called the *update gate*, which acts as coupled input and forget gate. This idea was described earlier in Equation (3.30). The activations $\boldsymbol{r}(t)$ are called *reset gate*. It regulates how strongly the increments $\boldsymbol{z}(t)$ depend on the old hiddens $\boldsymbol{h}(t-1)$.

3.7.1 Backpropagation for GRU

The gradients with respect to the hiddens are

$$\begin{aligned}
\frac{\partial L}{\partial \boldsymbol{h}(t)} =& \frac{\partial L(t)}{\partial \boldsymbol{h}(t)} \\
&+ \frac{\partial L}{\partial \boldsymbol{h}(t+1)} \frac{\partial \boldsymbol{h}(t+1)}{\partial \boldsymbol{h}(t)} \\
&+ \frac{\partial L}{\partial \boldsymbol{i}(t+1)} \frac{\partial \boldsymbol{i}(t+1)}{\partial \boldsymbol{h}(t)} \\
&+ \frac{\partial L}{\partial \boldsymbol{r}(t+1)} \frac{\partial \boldsymbol{r}(t+1)}{\partial \boldsymbol{h}(t)} \\
&+ \frac{\partial L}{\partial \boldsymbol{z}(t+1)} \frac{\partial \boldsymbol{z}(t+1)}{\partial \boldsymbol{h}(t)} \\
=& \frac{\partial L(t)}{\partial \boldsymbol{h}(t)} \\
&+ \frac{\partial L}{\partial \boldsymbol{h}(t+1)} \operatorname{diag}(\mathbf{1} - \boldsymbol{i}(t+1)) \\
&+ \frac{\partial L}{\partial \boldsymbol{i}(t+1)} \operatorname{diag}(\sigma'(\boldsymbol{R}^\top \boldsymbol{h}(t))) \boldsymbol{R}^\top \\
&+ \frac{\partial L}{\partial \boldsymbol{r}(t+1)} \operatorname{diag}(\sigma'(\boldsymbol{U}^\top \boldsymbol{h}(t))) \boldsymbol{U}^\top \\
&+ \frac{\partial L}{\partial \boldsymbol{z}(t+1)} \operatorname{diag}(\tanh'(\boldsymbol{s}(t+1))) \boldsymbol{V}^\top \operatorname{diag}(\boldsymbol{r}(t+1)),
\end{aligned} \tag{3.49}$$

where $\boldsymbol{s}$ is an auxiliary pre-activation vector defined as

$$\boldsymbol{s}(t) = \boldsymbol{W}^\top \boldsymbol{x}(t) + \boldsymbol{V}^\top (\boldsymbol{r}(t) \odot \boldsymbol{h}(t-1)). \tag{3.50}$$

The gradients with respect to the gate activations at time t are

$$\begin{aligned}
\frac{\partial L}{\partial \boldsymbol{i}(t)} &= \frac{\partial L}{\partial \boldsymbol{h}(t)} \frac{\partial \boldsymbol{h}(t)}{\partial \boldsymbol{i}(t)} = \frac{\partial L}{\partial \boldsymbol{h}(t)} \operatorname{diag}(\boldsymbol{z}(t) - \boldsymbol{h}(t-1)) \\
\frac{\partial L}{\partial \boldsymbol{z}(t)} &= \frac{\partial L}{\partial \boldsymbol{h}(t)} \frac{\partial \boldsymbol{h}(t)}{\partial \boldsymbol{z}(t)} = \frac{\partial L}{\partial \boldsymbol{h}(t)} \operatorname{diag}(\boldsymbol{i}(t)) \\
\frac{\partial L}{\partial \boldsymbol{r}(t)} &= \frac{\partial L}{\partial \boldsymbol{z}(t)} \frac{\partial \boldsymbol{z}(t)}{\partial \boldsymbol{r}(t)} = \frac{\partial L}{\partial \boldsymbol{z}(t)} \operatorname{diag}(\tanh'(\boldsymbol{s}(t))) \boldsymbol{V}^\top \operatorname{diag}(\boldsymbol{h}(t-1)).
\end{aligned} \tag{3.51}$$

And, similar to Equations (3.17-3.19) we define the deltas as:

$$\begin{aligned}
\delta \boldsymbol{i}(t) &= \left(\frac{\partial L}{\partial \boldsymbol{i}(t)} \frac{\partial \boldsymbol{i}(t)}{\partial (\boldsymbol{R}^\top \boldsymbol{h}(t-1))} \right)^\top = \left(\frac{\partial L}{\partial \boldsymbol{i}(t)} \operatorname{diag}(\sigma'(\boldsymbol{R}^\top \boldsymbol{h}(t-1))) \right)^\top \\
\delta \boldsymbol{r}(t) &= \left(\frac{\partial L}{\partial \boldsymbol{r}(t)} \frac{\partial \boldsymbol{r}(t)}{\partial (\boldsymbol{U}^\top \boldsymbol{h}(t-1))} \right)^\top = \left(\frac{\partial L}{\partial \boldsymbol{r}(t)} \operatorname{diag}(\sigma'(\boldsymbol{U}^\top \boldsymbol{h}(t-1))) \right)^\top \\
\delta \boldsymbol{z}(t) &= \left(\frac{\partial L}{\partial \boldsymbol{z}(t)} \frac{\partial \boldsymbol{z}(t)}{\partial \boldsymbol{s}(t)} \right)^\top = \left(\frac{\partial L}{\partial \boldsymbol{z}(t)} \operatorname{diag}(\tanh'(\boldsymbol{s}(t))) \right)^\top.
\end{aligned} \tag{3.52}$$

Finally, the gradients with respect to the weights are

$$\begin{aligned}
\frac{\partial L}{\partial \boldsymbol{R}} &= \sum_{t=1}^{T} \boldsymbol{h}(t-1)(\delta \boldsymbol{i}(t))^\top & \frac{\partial L}{\partial \boldsymbol{W}} &= \sum_{t=1}^{T} \boldsymbol{x}(t)(\delta \boldsymbol{z}(t))^\top \\
\frac{\partial L}{\partial \boldsymbol{U}} &= \sum_{t=1}^{T} \boldsymbol{h}(t-1)(\delta \boldsymbol{r}(t))^\top & \frac{\partial L}{\partial \boldsymbol{V}} &= \sum_{t=1}^{T} (\boldsymbol{r}(t) \odot \boldsymbol{h}(t-1))(\delta \boldsymbol{z}(t))^\top.
\end{aligned} \tag{3.53}$$

3.7.2 Usage

GRUs are often used for problems where they yield (or are expected to yield) a similar performance to LSTMs and a simpler structure is preferable. As an application example for the GRU let us look at *statistical machine translation*, the original use-case given by Cho et al. (2014). Statistical machine translation (SMT) is a translation approach where a translation is made by using a (statistical) model derived from the analysis of bilingual text corpora. Other traditionally used approaches for machine translation are for example the rule-based approaches.

For their experiments Cho et al. (2014) couple two GRUs in an *Encoder-Decoder* fashion (see Figure 3.24): The first GRU is the encoder. It gets a tokens form a sentence in language 1, say German, and generates a latent representation z from it. Concretely, Cho et al. (2014) used the last hidden states of the encoder as representation. The so obtained representation is then concatenated to the input of the decoder GRU, which returns tokens of a sentence in language 2, say English.

For the time of publication the GRU achieved high scores on the investigated tasks (expressed in terms of $BLEU$ score). Furthermore, the authors where able to qualitatively demonstrate that the latent space of the GRU captures linguistic regularities of the word- and the phrase-level (see: Figure 3.25).

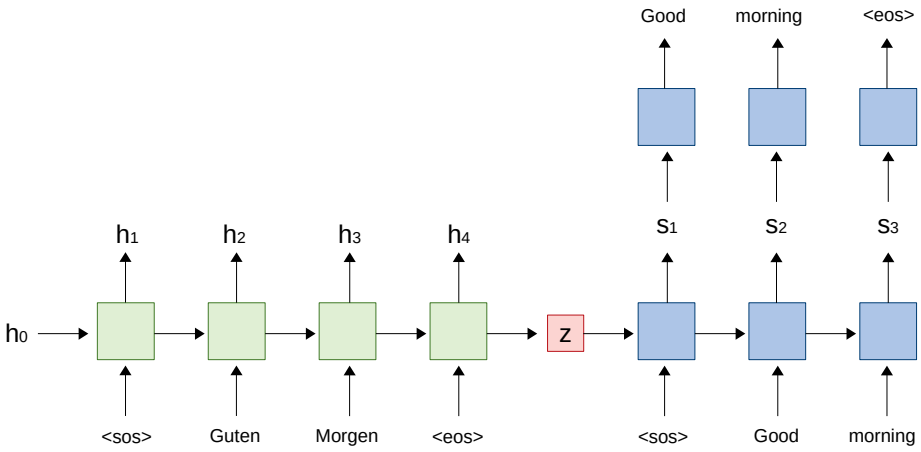


Figure 3.24: Schematic example of the GRU based architecture for machine driven language translation as proposed by Cho et al. (2014). The green boxes mark the encoder, the red one the obtained latent representation and the blue ones the decoder. Note that the output of the encoder is not used directly.

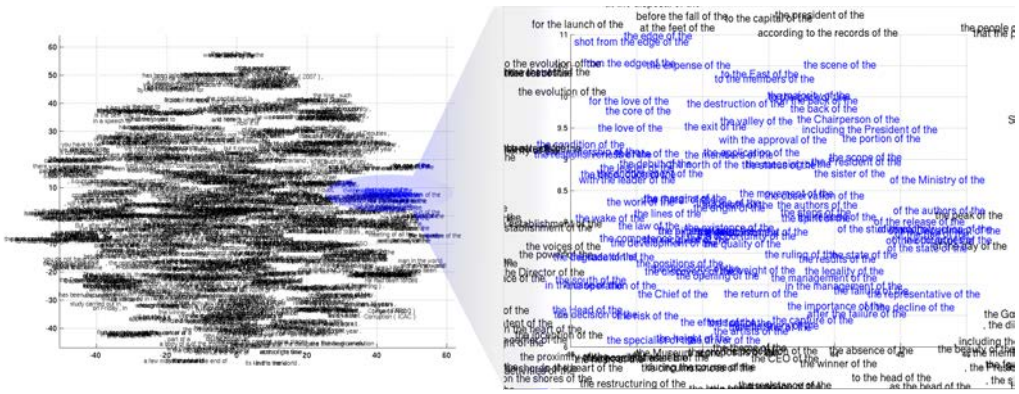


Figure 3.25: Projected 2-D word embedding learned by the GRU (adapted from Cho et al., 2014).

Chapter 4

Transformers and Attention

4.1 Introduction

From our own experience we all have an intuitive understanding of what it means to pay attention (or to have a lack thereof). For example, if an object or person draws our attention to it, we are less aware of our other surrounding. In the "Moonwalking Bear Awareness Test" (see `http://www.awarenesstest.co.uk/moonwalking-bear-awareness-test/`), you have to watch a video carefully and count the number of basketball passes by the players in white shirts. Many people do not spot the moonwalking bear that is crossing the field. Attention allows humans to efficiently use their limited resources to solve a task by selecting information that is relevant for the task while ignoring task-irrelevant information. Therefore complex tasks can be performed with limited resources for perception and information processing. More precise, attention "...refers to the process by which organisms select a subset of available information upon which to focus for enhanced processing [...] and integration" (from `http://www.scholarpedia.org/article/Attention`).

4.1.1 Spatial Attention

While the focus of this lecture is on temporal attention, we start with spatial attention. Spacial attention is to focus on relevant regions of an image when processing the image.

To distinguish an image of a cat from an image of a dog, not every detail of the image needs to be processed. Translating a long, complicated sentence might need attention on different words or different parts of the sentence at a time. To find an appropriate caption for an image, a distinction between important and dispensable content has to be made. As an example, Figure 4.1 depicts the latter scenario that was tackled by Xu et al. (2015a,b). The images show some captions produced by a neural network and the corresponding input images. On the right hand side of each input image, a attention map is depicted which shows to which parts of the image the network learned to attend.

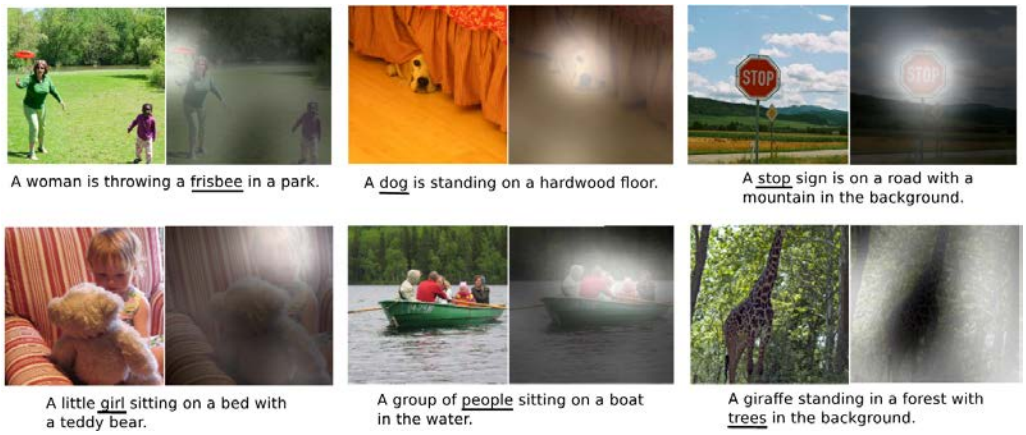


Figure 4.1: A example of an attention mechanism for image captioning. The bright parts of the right hand side images indicate, where the network focuses its attention to generate the caption. Taken from Xu et al. (2015a).

Another example of an spatial attention mechanism is the *Deep Recurrent Attentive Writer* (DRAW) neural network architecture that uses an attention mechanism for drawing Gregor et al. (2015a,b). In Figure 4.2 the red rectangles indicate where and to what extend the network pays attention. In Figure 4.3 explains how the filters work by zooming into an image. The goal is to classify MNIST digits. But unlike in the original MNIST data set, the images are distorted with clutters and translated i.e. they are not located in the center of the image. The difficulty for the network lies in finding the relevant information in the image. The DRAW network is an LSTM network that repeatedly looks an image until it has identified the relevant information. This is depicted in Figure 4.4a for various samples.

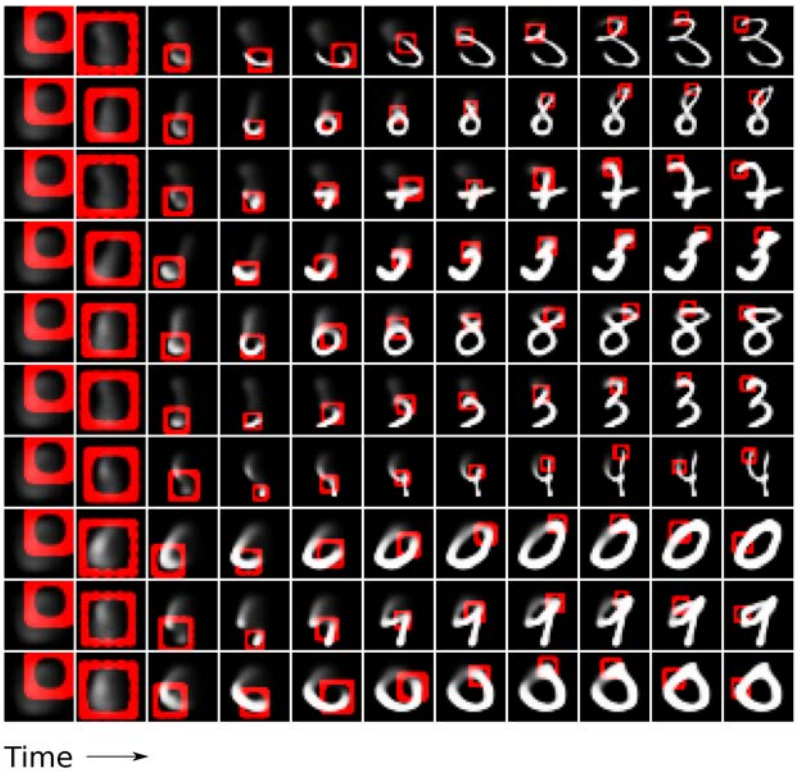


Figure 4.2: A trained DRAW network generating MNIST digits. Each row shows successive stages in the generation of a single digit. The red rectangle delimits the area attended to by the network at each time step, with the focal precision indicated by the width of the rectangle border. Taken from Gregor et al. (2015b).

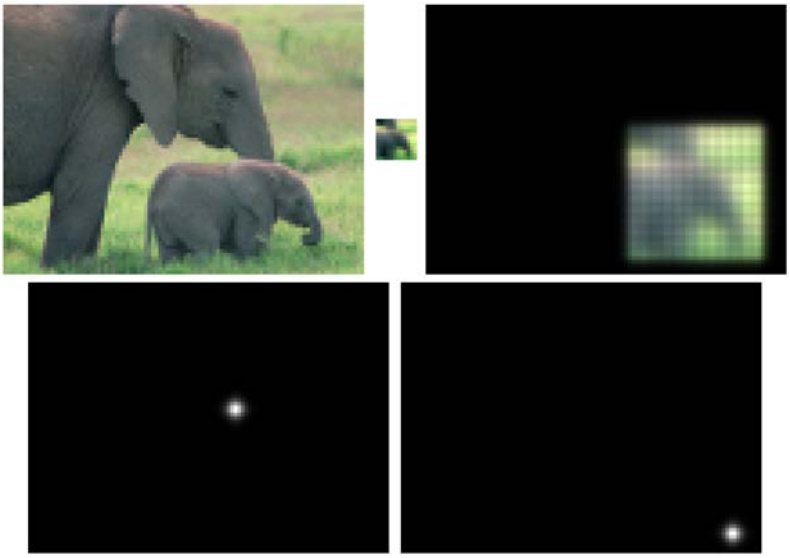
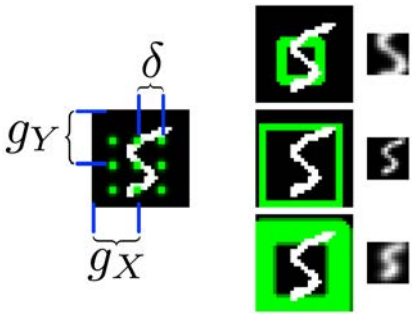


Figure 4.3: Zooming. Top Left: The original 100×75 image. Top Middle: A 12×12 patch extracted with 144 2D Gaussian filters. Top Right: The reconstructed image when applying transposed filters on the patch. Bottom: Only two 2D Gaussian filters are displayed. The first one is used to produce the top-left patch feature. The last filter is used to produce the bottom-right patch feature. By using different filter weights, the attention can be moved to a different location. Taken from Gregor et al. (2015b).

The idea of the mechanism is to apply an $N \times N$ grid of 2D Gaussian filters to an input image $\boldsymbol{x} \in \mathbb{R}^{H \times W}$, where H and W are the height and with of the image. Figure 4.4b depicts such a grid consisting out of nine 2D Gaussians. The mechanism uses isotropic Gaussians with the same variance σ^2 for all of the filters. The grid center is determined by the tuple (g_X, g_Y), the strides between the centers by δ.



(a) Each sequence shows a succession of four glimpses taken by the network while classifying cluttered translated MNIST. The green rectangle indicates the size and location of the attention patch, while the line width represents the variance of the filters. Taken from Gregor et al. (2015b).

(b) A 3×3 grid of filters superimposed on an image. The stride (δ) and centre location (g_X, g_Y) are indicated. Right: Three $N \times N$ patches extracted from the image. Taken from Gregor et al. (2015b).

In the following we will focus on **temporal attention** as it is very successful in language

processing like translation. The concepts of temporal attention can, however, transferred to spatial attention. We start with an overview over the most important concepts of temporal neural network attention. However, attention is a rapidly evolving field within machine learning whose scope goes far beyond this lecture. In line with the topic of this lecture we will focus on temporal attention.

4.1.2 Gates Introduced Temporal Attention

In contrast to spatial attention for images, temporal attention for sequences focuses on relevant elements or intervals of a sequence when processing this sequence.

Without explicitly mentioning it, we already have dealt with an attention mechanisms. The gating mechanisms of LSTM memory cells have the purpose of controlling which information is used or kept and which information is ignored. Gates can decide which information enters a memory cell, which information is kept over time, and which information can be disregarded or scaled down. Gating is focusing on a subset of the available information, i.e. is an attention mechanism. The LSTM gates already forestall the basic mechanisms of attention. As an example consider the input gate $\boldsymbol{i}(t)$ of an LSTM memory cell:

$$\boldsymbol{i}(t) = \sigma(\boldsymbol{W}_i^\top \boldsymbol{x}(t) + \boldsymbol{R}_i^\top \boldsymbol{y}(t-1)) \tag{4.1}$$

$$\boldsymbol{z}(t) = g(\boldsymbol{W}_z^\top \boldsymbol{x}(t) + \boldsymbol{R}_z^\top \boldsymbol{y}(t-1)) \tag{4.2}$$

$$\boldsymbol{c}(t) = \boldsymbol{c}(t-1) + \boldsymbol{i}(t) \odot \boldsymbol{z}(t). \tag{4.3}$$

Since the input gate activation vector is a vector of the form $\boldsymbol{i}(t) \in (0,1)^I$, the element-wise multiplication with the cell input activation in Eq. (4.2) controls which, and to which degree, elements of $\boldsymbol{z}(t)$ are used to update the cell state. The elements of $\boldsymbol{i}(t)$ can be interpreted as the amount of attention every element of $\boldsymbol{z}(t)$ receives. For the vanilla LSTM the attention $\boldsymbol{i}(t)$ is based on the same information that is used to calculate the cell input activation $\boldsymbol{z}(t)$, namely $\boldsymbol{x}(t)$ and $\boldsymbol{y}(t-1)$. The difference in the activation of $\boldsymbol{i}(t)$ and $\boldsymbol{z}(t)$ lies merely in the information that is extracted and used. However in the focused LSTM, gates only use $\boldsymbol{y}(t-1)$ and $\boldsymbol{z}(t)$ only uses $\boldsymbol{x}(t)$.

Consider a particular element $\boldsymbol{i}_k(t)$ of the input gate activation vector. The time step at t is fixed, therefore we denote $\boldsymbol{i}_k(t)$ by $\boldsymbol{i}_k$. Further we will ignore the recurrent part of Eq. (4.1). The k-th element of the input gate activation vector is

$$\boldsymbol{i}_k = \sigma(\sum_{l=1}^{d} \boldsymbol{w}_{lk}\boldsymbol{x}_l), \tag{4.4}$$

where d is the dimension of our input vector: $\boldsymbol{x} \in \mathbb{R}^d$. In the vocabulary that has developed with respect to attention mechanisms in deep learning, we would refer to the inner product between the $k-th$ column of $\boldsymbol{W}_i$ and the input vector as the *attention score* $e_k \in \mathbb{R}$:

$$e_k = \sum_{l=1}^{d} \boldsymbol{w}_{lk}\boldsymbol{x}_l \; = \boldsymbol{w}_{*k}^\top \boldsymbol{x} \tag{4.5}$$

$$= \|\boldsymbol{w}_{*k}\| \, \|\boldsymbol{x}\| \, \cos(\angle(\boldsymbol{w}_{*k}, \boldsymbol{x})). \tag{4.6}$$

From Eq. (4.6) we see that the magnitude of e_k depends on the product of the norms of $\boldsymbol{w}_{*k}$ and $\boldsymbol{x}$ while the sign depends on their angle. The interpretation is straight forward. If they point

into a similar direction and the product of magnitudes is high, the attention will be high because $\sigma(e_k) \approx 1$. If the two vectors are approximately perpendicular to each other, the inner product will be close to zero and thus $\sigma(e_k) \approx 0.5$. And if they point into opposing directions with high product of magnitudes, the attention will be close to zero. This way, every column of the weight matrix $\boldsymbol{W}_i$ can learn to selectively choose input vectors with certain traits. Every entry of the input gate from Section 4.1.2 is of the form $\boldsymbol{i}_k \in (0, 1)$. That can be considered as i_k paying *soft attention* to the entry $\boldsymbol{z}_k$, since it can also choose to let through half of the information. Would the gate be either zero or one, i.e. $\boldsymbol{i}_k \in \{0, 1\}$, the gate i_k is paying *hard attention* to the entry $\boldsymbol{z}_k$.

4.1.3 Temporal Attention in Artificial Neural Networks

All attention mechanisms in artificial neural networks have in common that they can decide which information is used for solving a task and which information is not used for solving a task. This decision may occur at the network input or internally. For example, an attention mechanism can be directly applied at the input image of a convolutional network. As mentioned above, LSTM applies attention to some internal activations of the network. We introduce the main ingredients of attention mechanisms on an abstract level, in order to relate different attention architectures to one another. Later, specialized attention mechanisms are introduced and explained, where their functions and modules will be linked to the concepts that are introduced here.

We want to apply an attention mechanism on the vector $\boldsymbol{z} \in \mathbb{R}^v$. For example, this could be the input image of an CNN, or it could be hidden states of a neural network like in Section 4.1.2 the cell input activation of an LSTM. In general, $\boldsymbol{z}$ can be expressed as the output of a function f_θ of a input vector $\boldsymbol{x} \in \mathbb{R}^d$:

$$\boldsymbol{z} = f_\theta(\boldsymbol{x}) \tag{4.7}$$

An attention mechanism directly applied to the input would lead to f_θ being the identity function and $\boldsymbol{z} = \boldsymbol{x}$. In the example from Section 4.1.2 it would be the cell input activation $\boldsymbol{z} = g(\boldsymbol{W}_z^\top \boldsymbol{x})$, where again the recurrent part is ignored.

Choosing the important information from $\boldsymbol{z}$ is realized by a function f_{att} and results in the attention vector $\boldsymbol{a} \in [0, 1]^v$. Then the attention vector $\boldsymbol{a}$ is applied to $\boldsymbol{z}$ via a convolution function f_{con} which results in the so-called *glimpse* or *context vector* $\boldsymbol{c}$:

$$\boldsymbol{a} = f_{\text{att}}(\boldsymbol{x}), \tag{4.8}$$

$$\boldsymbol{c} = f_{\text{con}}(\boldsymbol{a}, \boldsymbol{z}). \tag{4.9}$$

In the LSTM example the attention function would be the input gate: $\boldsymbol{a} = f_{\text{att}}(x) \equiv \sigma(\boldsymbol{W}_i^\top \boldsymbol{x}(t))$, and the glimpse would be the element wise product of the input gate and the cell input activation: $\boldsymbol{c} = f_{\text{con}}(\boldsymbol{i}, \boldsymbol{z}) \equiv \boldsymbol{i} \odot \boldsymbol{z}$. Often f_{con} is simply the Hadamard product between the vectors $\boldsymbol{a}$ and $\boldsymbol{z}$. The attention function f_{att} itself is a function composition of two functions, namely the normalizing function h and the scoring function f_{score}:

$$f_{\text{att}} = h \circ f_{\text{score}}. \tag{4.10}$$

Often the normalizing funciton h is the softmax, but other functions are used as well. In the case of our LSTM-example h was the element-wise sigmoid function $h(\cdot) \equiv \sigma(\cdot)$. The scoring function

f_{score} is a defining part of many attention mechanisms. As seen later, many mechanisms only differ by f_{score}.

While this section tries to give a general view on how attention is implemented in neural networks, it is inevitably a very abstract description. The realizations vary considerably, which is especially true for f_{att} as well as f_{con}. Next we will introduce various models that implement attention mechanisms.

4.2 Attention in Sequence-To-Sequence Models

As described in Section 3.5.1, the basic seq2seq model consists of two RNN networks, namely the encoder and the decoder network. The encoder reads a sentence $(\boldsymbol{x}(t))_{t=1}^{T}$ of length T from the source language and produces a fixed-dimensional vector representation $\boldsymbol{v}$ of the whole sentence. The encoder uses an implicit attention mechanism when processing the input sequence, as it can store relevant information in $\boldsymbol{v}$ but disregard irrelevant information. Typically $\boldsymbol{v}$ is the last state $\boldsymbol{h}_T$ of the encoder network but can also be the last cell state $\boldsymbol{c}_T$. This representation is then used to initialize the hidden states of the decoder. The model is depicted in Figure 4.5. Based on this information the encoder should compute the conditional probability of the target sequence $(\boldsymbol{y}(t))_{t=1}^{T'}$ of length T' being a translation of the input sequence or a proper text summarizing:

$$p(\boldsymbol{y}(1), \ldots, \boldsymbol{y}(T') \mid \boldsymbol{x}(1), \ldots, \boldsymbol{x}(T)) = \prod_{t=1}^{T} p(\boldsymbol{y}(t) \mid \boldsymbol{y}(1), \ldots, \boldsymbol{y}(t-1), \boldsymbol{v}) \tag{4.11}$$

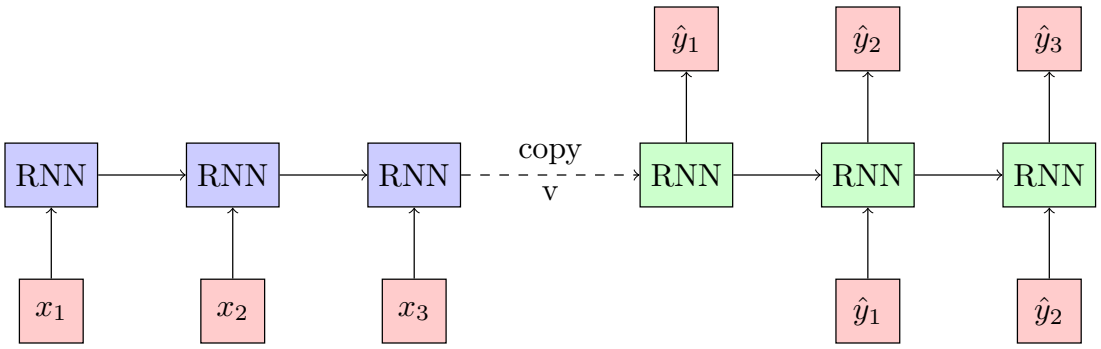


Figure 4.5: The encoder RNN (left, blue) learns a fixed-length representation of the input sequence. This vector is copied to the decoder RNN (right, green) which generates an output sequence based on the information contained in the encoded vector $\boldsymbol{v}$.

This encoder-decoder model was proposed by Cho et al. (2014) with GRUs. A similar but modified version that used LSTM was introduced by Sutskever et al. (2014). One obvious downside of this approach is the fact, that, no matter how long or complicated, all information of the input sequence needs to be encoded in one vector $\boldsymbol{v}$. As a human, one would most likely have a very different approach than reading the sentence once and than translating or summarizing it as a whole. One would instead look up passages that are important for the word that is translated.

The attention in this model is realized by the encoder RNN which is able to store relevant information in the input sequence or disregard information in the input sequence.

4.2.1 Additive Attention

One of the first attention mechanisms proposed by Bahdanau et al. (2014) allows to focus on different parts of the input sequence at a time. The encoder-decoder model combines attention with alignment.

The encoder consists of a bidirectional RNN (BiRNN) like described in Section 3.6.2. The forward RNN processes the input sequence $(\boldsymbol{x}(1), \ldots, \boldsymbol{x}(T))$ and produces forward hidden states $(\boldsymbol{h}_f(1), \ldots, \boldsymbol{h}_f(T))$. The backward RNN processes the input sequence in reverse order and creates the backward hidden states $(\boldsymbol{h}_b(1), \ldots, \boldsymbol{h}_b(T))$. At time step j, the hidden state is $\boldsymbol{h}(j) = [\boldsymbol{h}_f(j)^\top, \boldsymbol{h}_b(j)^\top]^\top$ and thus has information about the whole input sequence. The decoder RNN (DecRNN) generates at time step i the output $\boldsymbol{y}(i)$ based on the hidden state $\boldsymbol{s}(i)$. The hidden state is a function of the previous hidden state $\boldsymbol{s}(i-1)$, the output $\boldsymbol{y}(i-1)$, and a context vector $\boldsymbol{c}(i)$:

$$\boldsymbol{s}(i) = f(\boldsymbol{s}(i-1), \boldsymbol{y}(i-1), \boldsymbol{c}(i)). \tag{4.12}$$

The context vector $\boldsymbol{c}(i)$ is the result of the attention mechanism according to Section 4.1.3. For the calculation of the context vector $\boldsymbol{c}(i)$, an attention vector $\boldsymbol{a}$ is required as discussed in Section 4.1.3. First the attention score $\boldsymbol{e}(i)$ is computed for time step i with the j-th element $e_j(i)$. The attention score $e_j(i)$ is in principle an alignment score that determines how well the inputs around position j and the output at position i match. The score function is a feedforward neural network with parameters $\boldsymbol{W}$ and $\boldsymbol{U}$ and inputs $\boldsymbol{s}(i-1)$ and $\boldsymbol{h}(j)$. This feedforward network is jointly trained with the other RNNs and is:

$$e_j(i) = \boldsymbol{v}^\top \tanh(\boldsymbol{W}\boldsymbol{s}(i-1) + \boldsymbol{U}\boldsymbol{h}(j)), \tag{4.13}$$

where $\boldsymbol{W} \in \mathbb{R}^{n \times n}$, $\boldsymbol{U} \in \mathbb{R}^{n \times 2n}$ and $\boldsymbol{v} \in \mathbb{R}^n$ are the parameters. The dimension n is the size of the hidden vector in the BiRNN.

Attention mechanisms with score functions like in Eq. (4.13) are called *additive attention*. This refers to the sum inside the $\tanh$ function. Next the softmax function is applied over the attention score resulting in the attention vector $\boldsymbol{a}(i)$:

$$\boldsymbol{a}(i) = \mathrm{softmax}(\boldsymbol{e}(i)). \tag{4.14}$$

Now we can calculate the context vector $\boldsymbol{c}$, which is a weighted sum of the hidden representation of the encoding RNN:

$$\boldsymbol{c}(i) = \boldsymbol{H}\boldsymbol{a}(i) = \sum_{t=1}^{T} \boldsymbol{h}(t)\boldsymbol{a}_t(i) \tag{4.15}$$

where $\boldsymbol{H} = (\boldsymbol{h}(1), \ldots, \boldsymbol{h}(T))$ is a matrix with the hidden units as columns and thus f_{con} is simply the matrix-vector product of the attention vector and the hidden units. Figure 4.6 illustrates the proposed model that tries to generate the t-th target word $\boldsymbol{y}(t)$ given a source sentence $(\boldsymbol{x}(1) \ldots \boldsymbol{x}(T))$. Figure 4.7 shows a source sentence in English is translated into French. Every column is a attention vector.

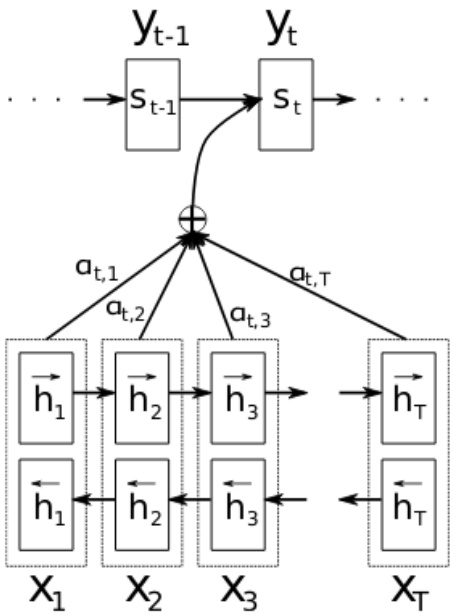


Figure 4.6: The graphical illustration of the proposed model trying to generate the t-th target word $\boldsymbol{y}(t)$ given a source sentence $(\boldsymbol{x}(1) \dots \boldsymbol{x}(T))$. Taken from Bahdanau et al. (2014)

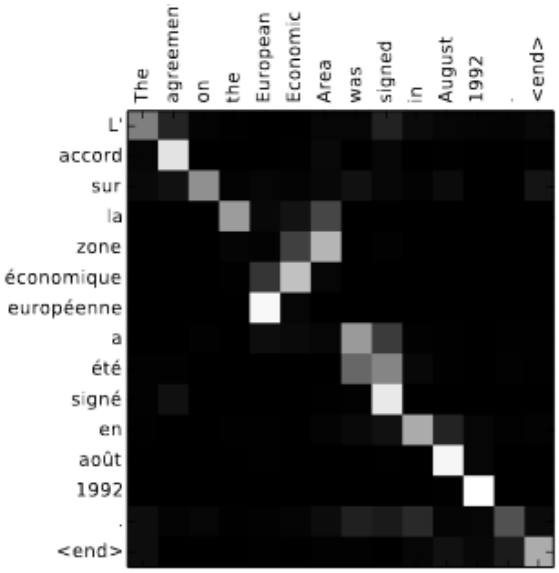


Figure 4.7: A source sentence in English is translated into French. Every column is a attention vector. Taken from Bahdanau et al. (2014)

4.2.2 Multiplicative Attention and Local Attention

The additive attention mechanisms proposed by Bahdanau et al. (2014) was modified and extended by Luong et al. (2015) to the multiplicative attention models and to local attention models. Instead

of a bidirectional GRU encoder multiplicative attention uses a unidirectional stacked LSTM encoder. Additional to the additive attention described in the previous section (called *concat* in their paper), the *dot* and *general* score function have been introduced. Combined they are referred to as *multiplicative attention*:

$$\boldsymbol{e}_t(i) = \boldsymbol{h}(t)^\top \boldsymbol{s}(i) \qquad \text{(dot)} \tag{4.16}$$

$$\boldsymbol{e}_t(i) = \boldsymbol{h}(t)^\top \boldsymbol{W} \boldsymbol{s}(i) \qquad \text{(general)} \tag{4.17}$$

$$\boldsymbol{e}_t(i) = \boldsymbol{v}^\top \tanh\left(\boldsymbol{W}\left[\boldsymbol{s}(i); \boldsymbol{h}(t)\right]\right) \qquad \text{(concat)}, \tag{4.18}$$

where the hidden states of the encoder network at time step t (in this case the last hiddens of the uppermost LSTM layer) are denoted by $\boldsymbol{h}(t)$. The hidden state of the decoder LSTM at time step i is denoted by $\boldsymbol{s}(i)$. $\boldsymbol{W}$ and $\boldsymbol{v}$ are learned parameters of the score function. The attention vector (in the paper called *alignment vector*) is calculated by applying the softmax function to the score

$$\boldsymbol{a}(i) = \mathrm{softmax}(\boldsymbol{e}(i)). \tag{4.19}$$

Multiplicative attention is simpler than additive attention. For short input sequences the performance of multiplicative attention is similar to that of additive attention, but for longer sequences additive attention is in advantage.

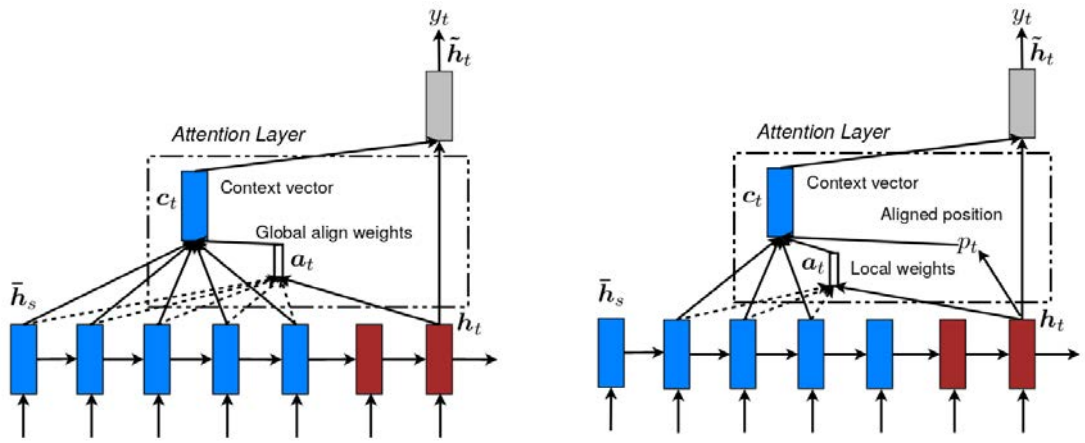
The described multiplicative attention mechanism attends to all hidden states of the last layer of the encoder network and thus is termed *global* attention. Luong et al. (2015) propose an extension of their mechanism called *local* attention. The idea is to not attend to all hidden units, but only to a subset of hidden units inside a window $[p(i) - D, p(i) + D], \quad D \in \mathbb{N}^+$ centered at position $p(i)$ at decoding time step i. Thus, the attention vector is of size $\boldsymbol{a}(i) \in \mathbb{R}^{2D+1}$. How to choose $p(i)$? Two strategies are proposed: *Monotonic Alignment* and *Predictive alignment*. Monotonic alignment simply assumes that the source and the target sequence are roughly monotonically aligned and thus it makes sense to set $p(i) = i$. Predictive alignment however is a attention mechanism at its own right. The position is calculated as follows:

$$p(i) = S\, \sigma(\boldsymbol{v}^\top \tanh(\boldsymbol{W}\boldsymbol{h}(i))) \tag{4.20}$$

where S is the source sentence length and $\sigma(\cdot)$ is the sigmoid function. Thus we have that $p(i) \in [0, S]$ denotes some position at the input sequence. The attention vector is calculated as follows:

$$\boldsymbol{a}_t(i) = \mathrm{softmax}(\boldsymbol{e}(i)) \exp\big(-\frac{(t - p(i))^2}{2\sigma^2}\big) \tag{4.21}$$

where the standard deviation is set to $\sigma = \frac{D}{2}$.



(a) (b)

Figure 4.8: Global (a) vs local (b) attention. Note, that the variable names in the paper differ from the names in this section. Taken from Luong et al. (2015).

4.2.3 Self-Attention

So far attention was based on the decoder having access to hidden states of the encoder. Cheng et al. (2016) introduced *intra-attention* or *self-attention* where an LSTM network has access to its own past cell states or hidden states. The idea is that all information about the previous states and inputs should be encoded in $\boldsymbol{h}(t)$. Therefore all relations between states should be induced, which is achieved by intra-attention or self-attention. Toward this end, Cheng et al. (2016) augment the LSTM architecture with a memory for the cell states $\boldsymbol{c}$ and the hidden states (LSTM outputs) $\boldsymbol{h}$. The memory just stores all past cell state vectors $\boldsymbol{c}$ in a matrix $\boldsymbol{C} = (\boldsymbol{c}(1), \ldots, \boldsymbol{c}(t-1))$ and all past hidden state vectors $\boldsymbol{h}$ in a matrix $\boldsymbol{H}(t-1) = (\boldsymbol{h}(1), \ldots, \boldsymbol{h}(t-1))$. Now the LSTM has direct access to its own past states, where the access mechanism is self-attention.

At time step t the attention score $\boldsymbol{e}(t)$ and attention vector $\boldsymbol{a}(t)$ are calculated over all previous time steps $i = 1, \ldots, t-1$:

$$e_i(t) = \boldsymbol{v}^\top \tanh(\boldsymbol{W}_h \boldsymbol{h}(i) + \boldsymbol{W}_x \boldsymbol{x}(t) + \boldsymbol{W}_{\tilde{\boldsymbol{h}}} \tilde{\boldsymbol{h}}(t-1)) \tag{4.22}$$

$$\boldsymbol{a}(t) = \mathrm{softmax}(\boldsymbol{e}(t)) \tag{4.23}$$

where for the t-th time step $\tilde{\boldsymbol{h}}(t)$ is calculated as

$$\tilde{\boldsymbol{h}}(t) = \boldsymbol{H}(t-1)\boldsymbol{a}(t-1) \tag{4.24}$$

and $\boldsymbol{H}(t-1) = (\boldsymbol{h}(1), \ldots, \boldsymbol{h}(t-1))$ is the matrix of all hidden states up to time step $t-1$. Also the attention vector is applied on all past cell states $\boldsymbol{C} = (\boldsymbol{c}(1), \ldots, \boldsymbol{c}(t-1))$:

$$\tilde{\boldsymbol{c}}(t) = \boldsymbol{C}(t-1)\boldsymbol{a}(t-1). \tag{4.25}$$

The LSTM forward pass is:

$$\boldsymbol{i}(t) = \sigma(\boldsymbol{W}_i^\top \boldsymbol{x}(t) + \boldsymbol{R}_i^\top \tilde{\boldsymbol{h}}(t)) \tag{4.26}$$
$$\boldsymbol{o}(t) = \sigma(\boldsymbol{W}_o^\top \boldsymbol{x}(t) + \boldsymbol{R}_o^\top \tilde{\boldsymbol{h}}(t)) \tag{4.27}$$
$$\boldsymbol{z}(t) = g(\boldsymbol{W}_z^\top \boldsymbol{x}(t) + \boldsymbol{R}_z^\top \tilde{\boldsymbol{h}}(t)) \tag{4.28}$$
$$\boldsymbol{c}(t) = \tilde{\boldsymbol{c}}(t) + \boldsymbol{i}(t) \odot \boldsymbol{z}(t) \tag{4.29}$$
$$\boldsymbol{h}(t) = \boldsymbol{o}(t) \odot \tanh(\boldsymbol{c}(t)). \tag{4.30}$$

The LSTM augmented with this memory and self-attention is called Long Short-Term Memory-Network (LSTMN).

4.2.4 Sentence Embedding via Self-Attention

In Section 3.5.1 it was explained how words can be embedded in to a d-dimensional real vector space $\mathbb{R}^d$. With the help of a self-attention mechanism, Lin et al. (2017) go a step further and embed whole sentences.

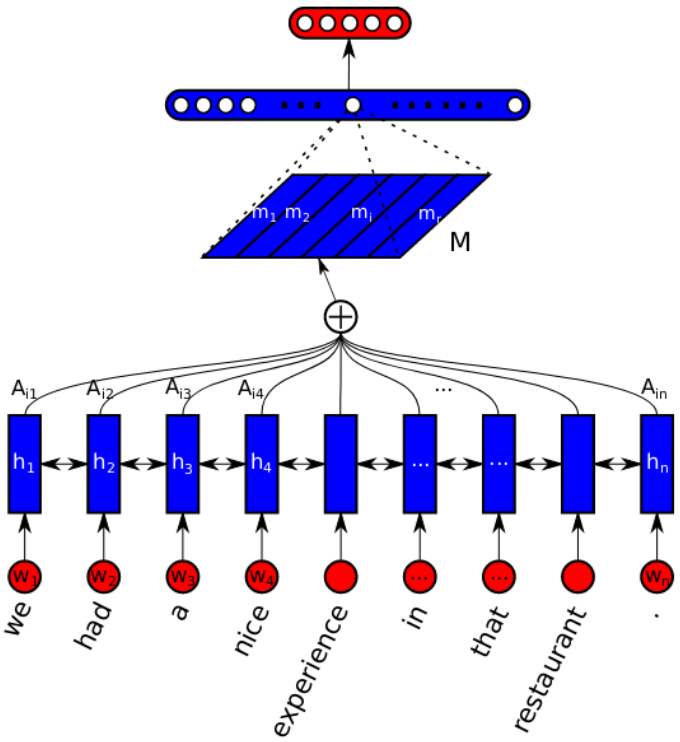


Figure 4.9: Embedding of whole sentences with self-attention according to Lin et al. (2017).

Like in Section 4.2.1, a bidirectional encoder realized via an LSTM. A sequence of words $\boldsymbol{x}(1), \ldots, \boldsymbol{x}(T)$ in $\mathbb{R}^d$ is embedded in $\mathbb{R}^d$. As previously, the forward hidden states $(\boldsymbol{h}_f(1), \ldots, \boldsymbol{h}_f(T))$ and the backward hidden states $(\boldsymbol{h}_b(1), \ldots, \boldsymbol{h}_b(T))$ are joint to a hidden state $\boldsymbol{h}(j) = [\boldsymbol{h}_f(j)^\top, \boldsymbol{h}_b(j)^\top]^\top$ for all $j = 1, \ldots, T$. For self-attention the hidden states are combined to the matrix

$$\boldsymbol{H} = (\boldsymbol{h}(1), \ldots, \boldsymbol{h}(T)) \in \mathbb{R}^{d \times T}. \tag{4.31}$$

The score function over the whole sentence is

$$\boldsymbol{e} = \boldsymbol{v}^\top \tanh(\boldsymbol{W}\boldsymbol{H}), \tag{4.32}$$

where $\boldsymbol{v} \in \mathbb{R}^v$ and $\boldsymbol{W} \in \mathbb{R}^{v \times d}$ are parameters, and the dimension $v \in \mathbb{N}^+$ can be freely chosen. The resulting attention vector can be applied to the matrix of hidden units to compute a context vector $\boldsymbol{c}$:

$$\boldsymbol{a} = \mathrm{softmax}(\boldsymbol{e}), \tag{4.33}$$
$$\boldsymbol{c} = \boldsymbol{H}\boldsymbol{a}. \tag{4.34}$$

Instead of just one context vector, multiple context vectors can be computed. Different context vectors might capture different aspects of the sentence. Towards this end, the vector $\boldsymbol{v} \in \mathbb{R}^v$ is replaced by a matrix $\boldsymbol{V} \in \mathbb{R}^{m \times v}$, where $m \in \mathbb{N}^+$ is the number of context vectors. The score than is calculated as

$$\boldsymbol{E} = \boldsymbol{V} \tanh(\boldsymbol{W}\boldsymbol{H}) \in \mathbb{R}^{m \times T}, \tag{4.35}$$

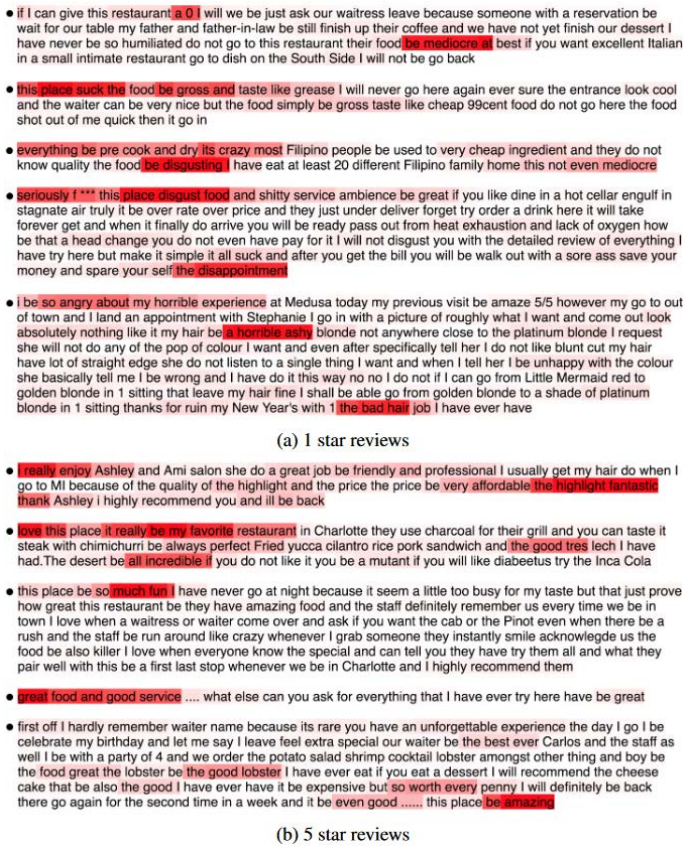
and the sentence embedding is calculated as

$$\boldsymbol{A} = \mathrm{softmax}(\boldsymbol{E}), \tag{4.36}$$
$$\boldsymbol{C} = \boldsymbol{H}\boldsymbol{A}^\top \in \mathbb{R}^{d \times m}. \tag{4.37}$$

The context matrix only is of advantage if individual context vectors hold different information, i.e. the corresponding attention vectors attend to different hidden vectors. Variety in context vectors is enforced by aiming at attention vectors that are mutually orthogonal. An objective for orthogonality is the following term that have to be minimized:

$$R = \|\boldsymbol{A}\boldsymbol{A}^T - \boldsymbol{I}\|_F, \tag{4.38}$$

where $\boldsymbol{I}$ is the identity matrix and $\|.\|_F$ the Frobenius norm. Figure 4.9 depicts schematically a network with an embedding layer. The embedding matrix is denoted as as $\boldsymbol{M}$ and a sentiment classification network is built on top of the embedding.



- if I can give this restaurant a 0 I will we be just ask our waitress leave because someone with a reservation be wait for our table my father and father-in-law be still finish up their coffee and we have not yet finish our dessert I have never be so humiliated do not go to this restaurant their food be mediocre at best if you want excellent Italian in a small intimate restaurant go to dish on the South Side I will not be go back

- this place suck the food be gross and taste like grease I will never go here again ever sure the entrance look cool and the waiter can be very nice but the food simply be gross taste like cheap 99cent food do not go here the food shot out of me quick then it go in

- everything be pre cook and dry its crazy most Filipino people be used to very cheap ingredient and they do not know quality the food be disgusting I have eat at least 20 different Filipino family home this not even mediocre

- seriously f *** this place disgust food and shitty service ambience be great if you like dine in a hot cellar engulf in stagnate air truly it be over rate over price and they just under deliver forget try order a drink here it will take forever get and when it finally do arrive you will be ready pass out from heat exhaustion and lack of oxygen how be that a head change you do not even have pay for it I will not disgust you with the detailed review of everything I have try here but make it simple it all suck and after you get the bill you will be walk out with a sore ass save your money and spare your self the disappointment

- i be so angry about my horrible experience at Medusa today my previous visit be amaze 5/5 however my go to out of town and I land an appointment with Stephanie I go in with a picture of roughly what I want and come out look absolutely nothing like it my hair be a horrible ashy blonde not anywhere close to the platinum blonde I request she will not do any of the pop of colour I want and even after specifically tell her I do not like blunt cut my hair have lot of straight edge she do not listen to a single thing I want and when I tell her I be unhappy with the colour she basically tell me I be wrong and I have do it this way no no I do not if I can go from Little Mermaid red to golden blonde in 1 sitting that leave my hair fine I shall be able go from golden blonde to a shade of platinum blonde in 1 sitting thanks for ruin my New Year's with 1 the bad hair job I have ever have

(a) 1 star reviews

- I really enjoy Ashley and Ami salon she do a great job be friendly and professional I usually get my hair do when I go to MI because of the quality of the highlight and the price the price be very affordable the highlight fantastic thank Ashley i highly recommend you and ill be back

- love this place it really be my favorite restaurant in Charlotte they use charcoal for their grill and you can taste it steak with chimichurri be always perfect Fried yucca cilantro rice pork sandwich and the good tres lech I have had.The desert be all incredible if you do not like it you be a mutant if you will like diabeetus try the Inca Cola

- this place be so much fun I have never go at night because it seem a little too busy for my taste but that just prove how great this restaurant be they have amazing food and the staff definitely remember us every time we be in town I love when a waitress or waiter come over and ask if you want the cab or the Pinot even when there be a rush and the staff be run around like crazy whenever I grab someone they instantly smile acknowlegde us the food be also killer I love when everyone know the special and can tell you they have try them all and what they pair well with this be a first last stop whenever we be in Charlotte and I highly recommend them

- great food and good service .... what else can you ask for everything that I have ever try here have be great

- first off I hardly remember waiter name because its rare you have an unforgettable experience the day I go I be celebrate my birthday and let me say I leave feel extra special our waiter be the best ever Carlos and the staff as well I be with a party of 4 and we order the potato salad shrimp cocktail lobster amongst other thing and boy be the food great the lobster be the good lobster I have ever eat if you eat a dessert I will recommend the cheese cake that be also the good I have ever have it be expensive but so worth every penny I will definitely be back there go again for the second time in a week and it be even good ..... this place be amazing

(b) 5 star reviews

Figure 4.10: Sentiment analysis of Yelp reviews based on sentence embedding. Taken from Lin et al. (2017)

4.3 Key-Value Attention

In Section 4.1.3 an overview over the main ingredients of most attention mechanisms was given. Three main steps where outlined: The calculation of a attention score $\boldsymbol{e}$, the transformation of the score to an attention vector $\boldsymbol{a}$ and the calculation of the context vector or glimpse $\boldsymbol{c}$. Daniluk et al. (2017) analyzed, that in this setting, the hidden states have multiple roles to fulfill, namely that they

- encode a distribution for predicting the next token,
- serve as a key to compute the attention vector,
- encode relevant content to inform future predictions.

We will describe how the key-value mechanism circumvents the problem of those multiple roles. In fact, the introduction of this mechanism was very influential and lead to the so called *Transformer* architecture (see next section), which made it possible to process sequences without using RNNs.

The main idea of Daniluk et al. (2017) is to encode the different roles directly into the hidden states. Namely, a hidden vector is separated into a *Key* part $\boldsymbol{k}$ and into a value part $\boldsymbol{v}$, both of size d:

$$\boldsymbol{h}(t) = (\boldsymbol{k}(t)^\top, \boldsymbol{v}(t)^\top)^\top \in \mathbb{R}^{2d}. \tag{4.39}$$

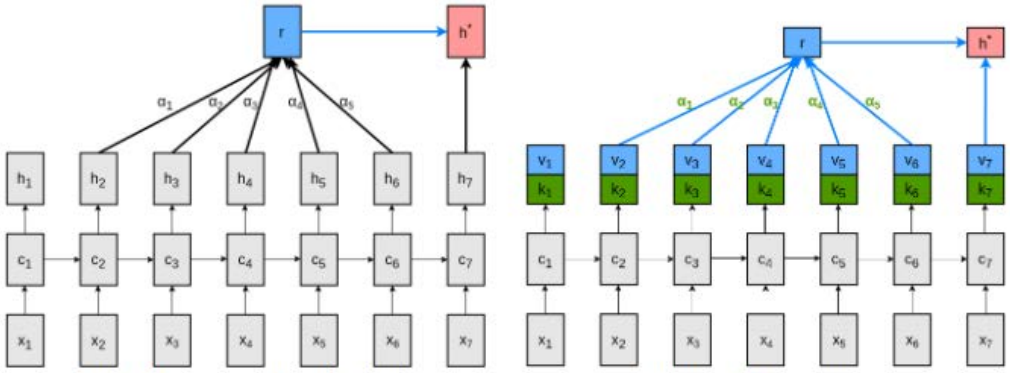
For the score at time step t, additive attention is applied on the keys in a sliding window of size L up to the current time step $\boldsymbol{K}(t-1) = (\boldsymbol{k}(t-L), \ldots, \boldsymbol{k}(t-1))$ and the current key $\boldsymbol{k}(t)$:

$$\boldsymbol{e}(t) = \boldsymbol{w}^\top \tanh(\boldsymbol{W}_K \boldsymbol{K} + (\boldsymbol{W}_k \boldsymbol{k}(t))\mathbf{1}^\top) \in \mathbb{R}^L, \tag{4.40}$$

where $\boldsymbol{W}_K, \boldsymbol{W}_k \in \mathbb{R}^{d\times d}$, $\boldsymbol{w} \in \mathbb{R}^d$ and $\mathbf{1} \in \mathbb{R}^L$. In the second summand we have a outer product that makes sure that $(\boldsymbol{W}_k \boldsymbol{k}(t))$
$BOn^\top \in \mathbb{R}^{d\times L}$. The attention vector is obtained by applying the softmax function to the score. The context vector is calculated by multiplying the attention vector by the values $\boldsymbol{V}(t-1) = (\boldsymbol{v}(t-L), \ldots, \boldsymbol{v}(t-1)) \in \mathbb{R}^{d\times L}$:

$$\boldsymbol{a}(t) = \operatorname{softmax}(\boldsymbol{e}(t)) \tag{4.41}$$

$$\boldsymbol{c}(t) = \boldsymbol{V}(t-1)\boldsymbol{a}(t) \quad \in \mathbb{R}^d \tag{4.42}$$



(a) Neural language model with attention. (b) Key-value separation.

Figure 4.11: Difference between previous attention mechanisms (a) and key-value attention (b). Here the context vector is denoted wit the letter $\boldsymbol{r}$ Taken from Daniluk et al. (2017)

4.3.1 Transformer Networks

End to end memory networks (EMN) **?** have become very popular since they were very successful in different applications.

The NeurIPS 2017 publication "Attention Is All You Need" from Google Brain introduced the Transformer which is based on EMN Vaswani et al. (2017b,a). It had great impact on the community. In contrast to the previous attention models, the Transformer is not an RNN but a feedforward neural network. The Transformer relies on an extension to the key-value attention, it uses the dot-product attention and is self-attending. Thus it combines previously introduced concepts.

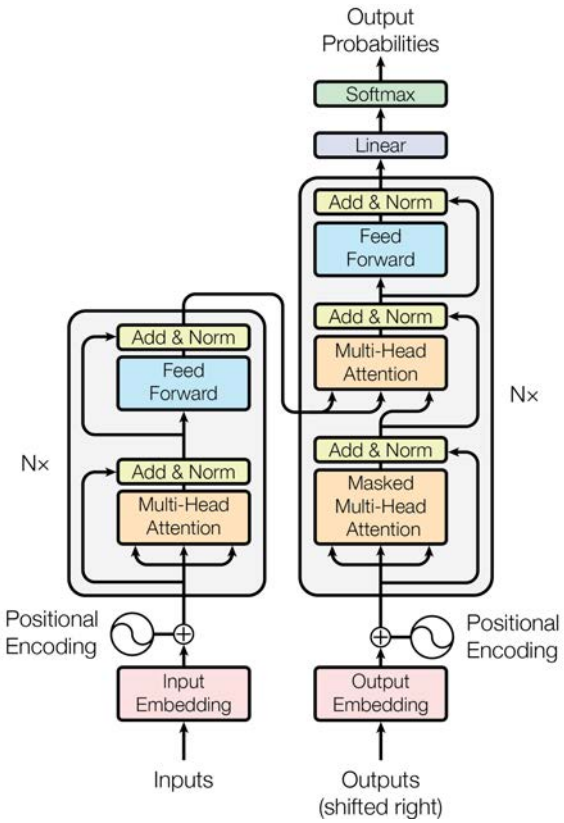


Figure 4.12: The Transformer model architecture. Left: a encoder module. Right: a decoder module. This modules are stacked N times. Taken from Vaswani et al. (2017a)

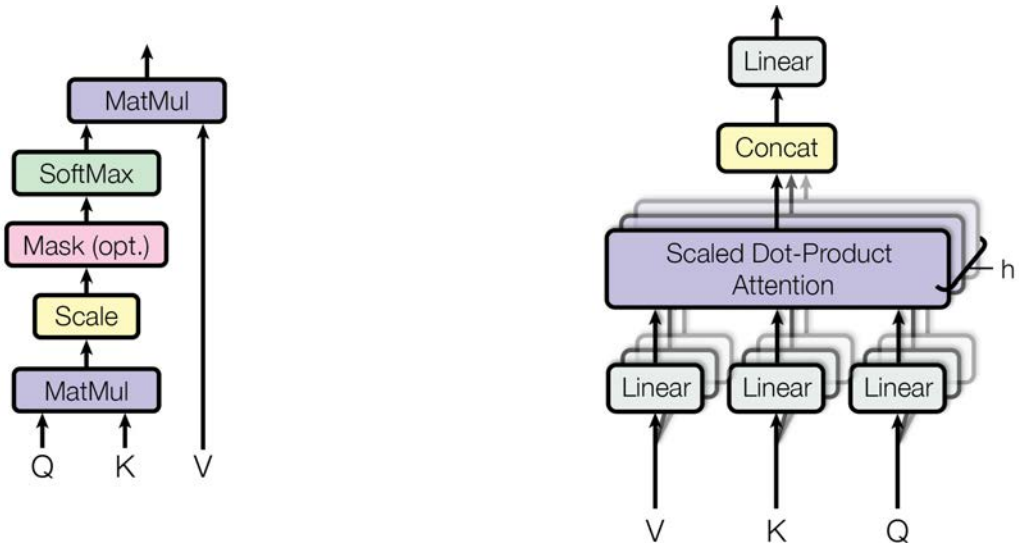


Figure 4.13: Scaled dot-product attention (left) and multihead attention (right). Taken from Vaswani et al. (2017a)

As depicted in Figure 4.12, the Transformer has a encoder-decoder architecture, where both,

the encoder and decoder modules are equipped with the so called *Multi-Head attention*. The input is a matrix of embedded words representing a sentence $\boldsymbol{X} = (\boldsymbol{x}(1), \ldots, \boldsymbol{x}(T)) \in \mathbb{R}^{d \times T}$. Since in a feedfoward network the temporal context is lost, a positional encoding $\boldsymbol{P}$ is added to the input. After that, Multi-Head attention is applied on the resulting output. The embedded input sentence with added positional encoding is denoted by $\boldsymbol{Z}_0 = \boldsymbol{X} + \boldsymbol{P} \in \mathbb{R}^{d \times T}$ and in general $\boldsymbol{Z}_n$ denotes the output of the n-th Transformer module. A Multi-head attention unit consist of a number of scaled dot-product attention units as depicted in Figure 4.13. Those units are composed of functionalities that were already introduced. The dot-product attention has been mentioned in Section 4.2.2. The key-value concept has been explained in the previous section. However, there are differences to the previous mechanisms. First the score function is treated. Additional to a key and a value, a term called *query* is introduced. All these components are calculated based on the input $\boldsymbol{Z}$. For the n-th stacked Transformer unit (i.e. layer), the query, the key, and the value are calculated as follows:

$$\boldsymbol{K}_n = \boldsymbol{W}_n^K \boldsymbol{Z}^{n-1} \in \mathbb{R}^{d_k}, \tag{4.43}$$

$$\boldsymbol{V}_n = \boldsymbol{W}_n^V \boldsymbol{Z}^{n-1} \in \mathbb{R}^{d_v}, \tag{4.44}$$

$$\boldsymbol{Q}_n = \boldsymbol{W}_n^Q \boldsymbol{Z}^{n-1} \in \mathbb{R}^{d_k}, \tag{4.45}$$

where the matrix subscript indicates the layer. The score, attention and context is calculated as

$$\boldsymbol{E}_n = \frac{1}{\sqrt{d_k}} \boldsymbol{Q}_n \boldsymbol{K}_n^\top, \tag{4.46}$$

$$\boldsymbol{A}_n = \mathrm{softmax}(\boldsymbol{E}_n), \tag{4.47}$$

$$\boldsymbol{C}_n = \boldsymbol{A}_n \boldsymbol{V}_n, \tag{4.48}$$

where the score is normalized with the square root of the key-dimension $\frac{1}{\sqrt{d_k}}$ to prevent the softmax function from saturation. Only one attention head has been described. However, Multi-Head attention consists of multiple such attention heads. Suppose we have h heads, together they produce h context matrices $\boldsymbol{C}_n^1, \ldots, \boldsymbol{C}_n^h$. The output of the Multi-Head attention unit than is calculated the following way:

$$\boldsymbol{R}_n = \mathrm{concat}(\boldsymbol{C}_n^1, \ldots, \boldsymbol{C}_n^h) \boldsymbol{W}_n^Z \in \mathbb{R}^{d \times T}. \tag{4.49}$$

The module is equipped with a shortcut connection that skips the Multi-Head unit. Therefore the Multi-Head attention unit output is $\tilde{\boldsymbol{R}}_n = \boldsymbol{R}_n + \boldsymbol{Z}_{n-1}$, which is normalized via a layer normalization. Thereafter the output is fed into a fully connected feedforward network. The resulting output is then passed to a Multi-Head attention unit in the decoder. The decoder is similar to the encoder but has two Multi-Head attention units. The first attention unit only receives input from previous decoder units and masks out subsequent positions of the input. The second attention unit also takes the output of the according encoding unit. In general, for a given sentence to translate, the model needs to encode the sentence once and than runs the decoder until a token arrives that indicates the end of the sequence.

4.3.2 BERT

Bidirectional Encoder Representations from Transformers (BERT) was introduced by Devlin et al. (2018, 2019). BERT is a language representation model based on the Transformer architecture.

In the pre-training phase it learns a language representation. The pre-training phase is similar to word embedding which was described earlier. Pre-training is done in a self-supervised fashion, that is, the labels are completely generated automatically. Specifically, BERT is pre-trained on two tasks. In the first *Mask LM task* the goal is to insert missing words into a sentence. Given is a sequence of embedded words $(\boldsymbol{x}(t))_{t=1}^{T}$ representing a sentence of length T. Words of this sentence are automatically masking out and the task is to insert the missing words. In the second *next sentence prediction* (NSP) task, BERT should learn to understand the relation between two sentences. The task is to decide whether a sentence is successor of another sentence or not.

Pre-training is very time consuming but must only be done once. Subsequently, the learned language representation can be leveraged for multiple purposes by fine-tuning the pre-trained model with respect to the specific task, like next sentence prediction or question answering.

Figure 4.14 shows the input representation of BERT. The input embeddings is sum of the token embeddings, the segmentation embeddings and the position embeddings. Figure 4.15 depicts BERT's overall pre-training and fine-tuning procedures. Apart from output layers, the same architectures are used in both pre-training and fine-tuning. During fine-tuning, all parameters are fine-tuned. [CLS] is a special symbol added in front of every input example, and [SEP] is a special separator token (e.g. separating questions / answers).

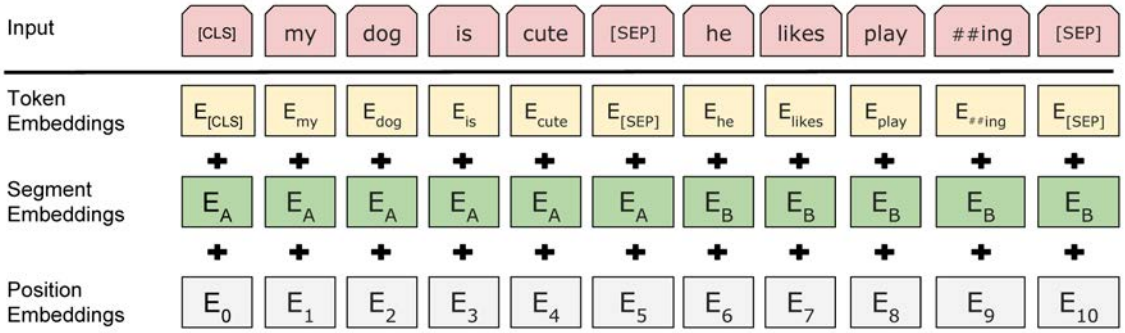


Figure 4.14: BERT input representation is sum of the token embeddings, the segmentation embeddings and the position embeddings. Taken from Devlin et al. (2019).

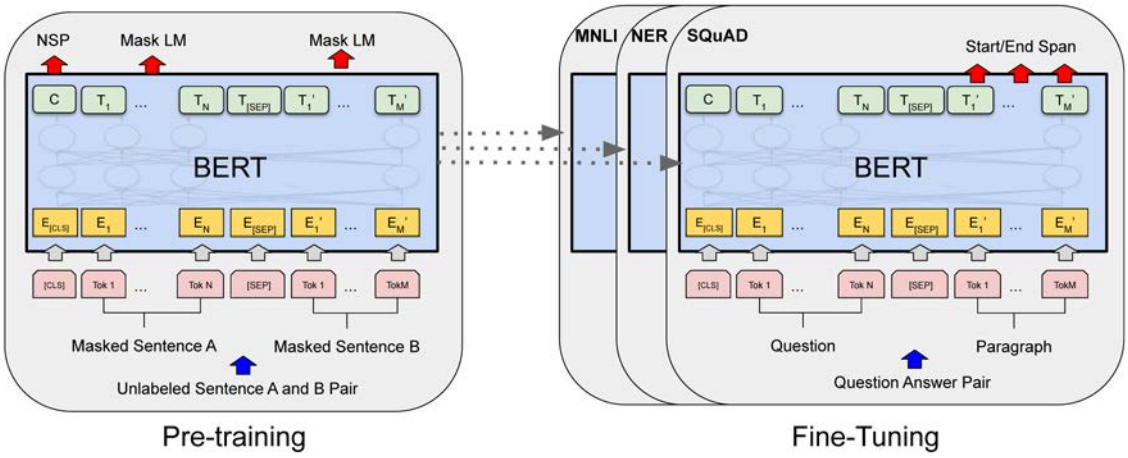


Figure 4.15: BERT pre-training and fine-tuning procedures. Taken from Devlin et al. (2019).

4.4 Advanced LSTM Architectures

4.4.1 Neural Turing Machine

4.4.2 Differentiable Neural Computer

4.4.3 Pointer Network

Chapter 5

Attractor Networks

Attractor networks constitute a class of neural networks that are very different to those we have considered so far. Nowadays, their practical relevance is rather low. However, they remain of historical and theoretical importance. Moreover, many attractor networks are biologically more plausible (than the networks described so far) because they apply *local learning rules*. Local means that the information needed to update a connection weight is located in the direct vicinity of that connection, i.e. the connection itself and the neurons attached to it.

Attractor networks are a class of recurrent neural networks that converge to special attractor points during the forward pass. Unlike RNNs we have considered so far, attractor networks do not operate on sequences. They process one point of their input space and then follow certain forward dynamics until they converge to one of their attractors. Attractors are stable points (or patterns) of the network's output space. Around an attractor there is a *basin of attraction*, which is defined to be the set of points from which the network will converge to that attractor. The forward dynamics of attractor networks are usually defined by weighted connections of their nodes (or neurons). The network dynamics may either be defined using continuous or discrete time.

Attractor networks can be used as associative memory. Given an input pattern, the network will converge to the closest output pattern (attractor) that the network has been trained to recognize.

5.1 Backpropagation for attractor networks with continuous time

Pineda (1987) demonstrated how attractor networks can be trained by backpropagation. Consider a network with units $\boldsymbol{x} \in \mathbb{R}^N$. Any of these units can be either an input unit, an output unit, or both, or none of it. In the latter case the unit is called a *hidden unit*. We denote the set of input units as A, and the set of output units as Ω. Note that $|A \cup \Omega| \leqslant N$ and that $A \cap \Omega$ may be nonempty. Input units receive source values $\alpha_i[x_i \in A]$ from the environment, where we used the Iverson brackets to set all source values for units $x_i \notin A$ to zero. The units are connected by a weight matrix $\boldsymbol{W} \in \mathbb{R}^{N \times N}$ and follow the continuous-time dynamics defined by the set of differential equations

$$\frac{\mathrm{d}x_i}{\mathrm{d}t} = -x_i + \sigma\left(\sum_{j=1}^{N} w_{ij} x_j\right) + \alpha_i[x_i \in A]. \tag{5.1}$$

Given these dynamics and a set of output values ω_i (which are arbitrary for non-output units), how do we have to adjust the weights such that the network activations will evolve to these values? Formally, we want to ensure that $\lim_{t\to\infty} x_i = x_i^\infty = \omega_i$ for all $x_i \in \Omega$.

Consider the energy function

$$E = \frac{1}{2}\sum_{i=1}^{N}\left([x_i \in \Omega](\omega_i - x_i^\infty)^2\right), \tag{5.2}$$

which is zero if all output units have converged to the attractor $\boldsymbol{\omega}$, and positive otherwise. Note that E depends on the weights through x_i^∞. We can now use continuous-time gradient descent to minimize E, that is we adjust the weights according to

$$\frac{\mathrm{d}w_{ij}}{\mathrm{d}t} = -\eta\frac{\partial E}{\partial w_{ij}} = \eta\sum_{k=1}^{N}\left([x_k \in \Omega](\omega_k - x_k^\infty)\frac{\partial x_k^\infty}{\partial w_{ij}}\right), \tag{5.3}$$

where η is the learning rate. Since the network dynamics (5.1) converge to x_k^∞, this must be a critical point of that system and hence a fixed point of the equation, i.e. it must satisfy $\mathrm{d}x_k/\mathrm{d}t = 0$, which implies

$$x_k^\infty = \sigma\left(\sum_{l=1}^{N} w_{kl}x_l^\infty\right) + \alpha_k[x_k \in A]. \tag{5.4}$$

Differentiating with respect to w_{ij} we get that

$$\begin{aligned}\frac{\partial x_k^\infty}{\partial w_{ij}} &= \sigma'\left(\sum_{l=1}^{N} w_{kl}x_l^\infty\right)\sum_{l=1}^{N}\left(x_l^\infty\frac{\partial w_{kl}}{\partial w_{ij}} + w_{kl}\frac{\partial x_l^\infty}{\partial w_{ij}}\right)\\ &= \sigma'\left(\sum_{l=1}^{N} w_{kl}x_l^\infty\right)\left(x_j^\infty[k=i] + \sum_{l=1}^{N} w_{kl}\frac{\partial x_l^\infty}{\partial w_{ij}}\right)\end{aligned} \tag{5.5}$$

by chain and product rule. To reduce notation, let us define the vector $\boldsymbol{u}$ by

$$u_i = \sum_{l=1}^{N} w_{il}x_l^\infty. \tag{5.6}$$

Now, rewriting Equation (5.5) in terms of $\boldsymbol{u}$ yields

$$\begin{aligned}\frac{\partial x_k^\infty}{\partial w_{ij}} &= \sigma'(u_k)\left(x_j^\infty[k=i] + \sum_{l=1}^{N} w_{kl}\frac{\partial x_l^\infty}{\partial w_{ij}}\right)\\ &= \sigma'(u_i)x_j^\infty[k=i] + \sum_{l=1}^{N}\sigma'(u_k)w_{kl}\frac{\partial x_l^\infty}{\partial w_{ij}},\end{aligned} \tag{5.7}$$

where the change from $\sigma'(u_k)$ to $\sigma'(u_i)$ has no effect because of the factor $[k=i]$. Further, let us define the matrix $\boldsymbol{A}$ as

$$a_{ij} = \sigma'(u_i)w_{ij} \tag{5.8}$$

and rewrite Equation (5.7) in terms of $\boldsymbol{A}$ and expand the recurrence relation, which gives

$$\begin{aligned}
\frac{\partial x_k^\infty}{\partial w_{ij}} &= \sigma'(u_i)x_j^\infty[k=i] + \sum_{l=1}^{N} a_{kl}\frac{\partial x_l^\infty}{\partial w_{ij}} \\
&= \sigma'(u_i)x_j^\infty[k=i] + \sum_{l=1}^{N} a_{kl}\left(\sigma'(u_i)x_j^\infty[l=i] + \sum_{m=1}^{N} a_{lm}\frac{\partial x_m^\infty}{\partial w_{ij}}\right) \\
&= \sigma'(u_i)x_j^\infty[k=i] + a_{ki}\sigma'(u_i)x_j^\infty + \sum_{l,m=1}^{N} a_{kl}a_{lm}\frac{\partial x_m^\infty}{\partial w_{ij}} \\
&= \sigma'(u_i)x_j^\infty[k=i] + a_{ki}\sigma'(u_i)x_j^\infty + \sum_{l,m=1}^{N} a_{kl}a_{lm}\left(\sigma'(u_i)x_j^\infty[m=i] + \sum_{n=1}^{N} a_{mn}\frac{\partial x_n^\infty}{\partial w_{ij}}\right) \\
&= \sigma'(u_i)x_j^\infty[k=i] + a_{ki}\sigma'(u_i)x_j^\infty + \sum_{l=1}^{N} a_{kl}a_{li}\sigma'(u_i)x_j^\infty + \sum_{l,m,n=1}^{N} a_{kl}a_{lm}a_{mn}\frac{\partial x_n^\infty}{\partial w_{ij}} \\
&= \sigma'(u_i)x_j^\infty[k=i] + a_{ki}\sigma'(u_i)x_j^\infty + \left(\boldsymbol{A}^2\right)_{ki}\sigma'(u_i)x_j^\infty + \sum_{l,m,n=1}^{N} a_{kl}a_{lm}a_{mn}\frac{\partial x_n^\infty}{\partial w_{ij}} \\
&= \sigma'(u_i)x_j^\infty\left(\boldsymbol{I}_{ki} + \boldsymbol{A}_{ki} + \left(\boldsymbol{A}^2\right)_{ki} + \left(\boldsymbol{A}^3\right)_{ki} + \cdots\right).
\end{aligned} \tag{5.9}$$

The expression in the brackets is the geometric series for the matrix $\boldsymbol{A}$, which has the limit $\lim_{n\to\infty}\sum_{p=0}^{n}\boldsymbol{A}^p = (\boldsymbol{I}-\boldsymbol{A})^{-1}$ if $\|\boldsymbol{A}\| < 1$ for any matrix norm $\|\cdot\|$. Using this fact we can write the gradient as

$$\frac{\partial x_k^\infty}{\partial w_{ij}} = \sigma'(u_i)x_j^\infty\left((\boldsymbol{I}-\boldsymbol{A})^{-1}\right)_{ki} \tag{5.10}$$

and plug it into Equation (5.3) to obtain the learning rule

$$\frac{\mathrm{d}w_{ij}}{\mathrm{d}t} = \eta x_j^\infty\sigma'(u_i)\sum_{k=1}^{N}[x_k\in\Omega](\omega_k - x_k^\infty)\left((\boldsymbol{I}-\boldsymbol{A})^{-1}\right)_{ki}. \tag{5.11}$$

This learning rule is non-local because of the matrix inversion. However, Pineda (1987) also gives an alternative formulation that is local.

5.2 Hopfield networks

An important example of attractor networks with discrete time are *Hopfield Networks* (Hopfield, 1982), which represent a form of *associative memory*. Trained with some example patterns, Hopfield Networks are able to reconstruct these patterns from corrupted versions of them. They are trained using the *Hebbian learning rule* (Hebb, 1949), which can be intuitively described as "cells that fire together, wire together," meaning that the connection between two neurons is strengthened when they tend to fire simultaneously. Therefore, Hopfield Networks are also of interest as a model for how human memory works.

Hopfield Networks consist of a binary state vector $\boldsymbol{x} \in \{-1, +1\}^D$, a bias vector $\boldsymbol{b} \in \mathbb{R}^D$, and a weight (or connectivity) matrix $\boldsymbol{W} \in \mathbb{R}^{D \times D}$ that fulfills $w_{ii} = 0$, i.e. cells have no self-connections, and $w_{ij} = w_{ji}$, i.e. connections are symmetric. On inference, given some initial state $\boldsymbol{x}$, the network follows the forward dynamics

$$\boldsymbol{x}^{\text{new}} = \text{sign}\left(\boldsymbol{W}\boldsymbol{x}^{\text{old}} - \boldsymbol{b}\right) \qquad \text{or} \qquad x_i^{\text{new}} = \text{sign}\left(\left(\sum_{j=1}^{D} w_{ij} x_j^{\text{old}}\right) - b_i\right) \tag{5.12}$$

equivalently. Now, if the weight w_{ij} is positive, the summand $w_{ij}x_j$ will have the same sign as x_j, i.e. the summand contributes into the direction of the value x_j. This can be interpreted as an attractive force between x_i and x_j. On the other hand, if w_{ij} is negative, then the summand $w_{ij}x_j$ contributes in the opposite direction, i.e. x_i and x_j repel each other. This update rule can be applied in either synchronous or asynchronous fashion, depending on whether all units are updated at once or one after another. Asynchronous updates are biologically more plausible as there is no indication of a global clock in the nervous system. Asynchronous updates can be applied in either fixed or random order of units.

Equation (5.12) minimizes the energy function E, which is defined as

$$E = -\frac{1}{2}\boldsymbol{x}^\top \boldsymbol{W}\boldsymbol{x} + \boldsymbol{x}^\top \boldsymbol{b} = -\frac{1}{2}\sum_{i=1}^{D}\sum_{j=1}^{D} w_{ij}x_i x_j + \sum_{i=1}^{D} b_i x_i. \tag{5.13}$$

This is the same definition of energy as used in the Ising model, a mathematical model of ferromagnetism. To see that Equation (5.12) minimizes the energy (5.13) locally, compare two subsequent energy levels

$$\begin{aligned} E^{\text{new}} - E^{\text{old}} =& -\frac{1}{2}\sum_{i=1}^{D}\sum_{j=1}^{D} w_{ij}x_i^{\text{new}}x_j^{\text{new}} + \sum_{i=1}^{D} b_i x_i^{\text{new}} \\ &+ \frac{1}{2}\sum_{i=1}^{D}\sum_{j=1}^{D} w_{ij}x_i^{\text{old}}x_j^{\text{old}} - \sum_{i=1}^{D} b_i x_i^{\text{old}} \\ =& -\frac{1}{2}\sum_{i=1}^{D}\sum_{j=1}^{D} w_{ij}\left(x_i^{\text{new}}x_j^{\text{new}} - x_i^{\text{old}}x_j^{\text{old}}\right) + \sum_{i=1}^{D} b_i\left(x_i^{\text{new}} - x_i^{\text{old}}\right). \end{aligned} \tag{5.14}$$

Now assume we updated only x_k, that is $x_i^{\text{new}} = x_i^{\text{old}}$ if $i \neq k$. Then only those terms of the double sum where either $i = k$ or $j = k$ are nonzero and remember that $w_{ij} = 0$ if $i = j$. We get

$$\begin{aligned} E_k^{\text{new}} - E_k^{\text{old}} =& -\sum_{i=1}^{D} w_{ik}\left(x_i^{\text{old}}x_k^{\text{new}} - x_i^{\text{old}}x_k^{\text{old}}\right) + b_k\left(x_k^{\text{new}} - x_k^{\text{old}}\right) \\ =& -\left(\sum_{i=1}^{D} w_{ik}x_i^{\text{old}}\right)\left(x_k^{\text{new}} - x_k^{\text{old}}\right) + b_k\left(x_k^{\text{new}} - x_k^{\text{old}}\right) \\ =& -\left(\sum_{i=1}^{D} w_{ik}x_i^{\text{old}} - b_k\right)\left(x_k^{\text{new}} - x_k^{\text{old}}\right). \end{aligned} \tag{5.15}$$

Now observe that the sign of the two expressions in brackets is always equal. If $x_k^{\text{new}} > x_k^{\text{old}}$ then the second expression is positive. Moreover, it implies that $x_k^{\text{new}} > 0$ which tells us that also the first expression must be positive by Equation (5.12). A similar argument covers the case $x_k^{\text{new}} < x_k^{\text{old}}$, where both expressions are negative. Either way, their product is positive and therefore $E^{\text{new}} - E^{\text{old}}$ is always nonpositive. Obviously, if $x_k^{\text{new}} = x_k^{\text{old}}$, the energy level remains unchanged because the network did not change. It follows that the energy can never increase when applying update rule (5.12) and therefore the network will converge to a local minimum of E.

However, it is important to note that not only stored patterns constitute local minima but the energy function can also have spurious local minima. Therefore, convergence to a local minimum does not imply that the network necessarily recovers one of the desired patterns.

New patterns can be trained to a Hopfield Network by adjusting the weights according to the Hebbian learning rule. Given N patterns $\{\boldsymbol{x}^{(n)}\}_{n=1}^{N}$, the weights are computed as a sum of outer products

$$\boldsymbol{W} = \frac{1}{N}\left(\sum_{n=1}^{N} \boldsymbol{x}^{(n)}\boldsymbol{x}^{(n)\top}\right) - \boldsymbol{I}_D. \tag{5.16}$$

Subtracting the identity matrix $\boldsymbol{I}_D$ ensures that the diagonal entries are zero, i.e. $w_{ii} = 0$. If the neurons $x_i^{(n)}$ and $x_j^{(n)}$ have the same sign, then this increases the weight w_{ij}, which in turn will introduce an attracting force between i-th and j-th neuron on inference as we have seen earlier. Likewise, if $x_i^{(n)}$ and $x_j^{(n)}$ have opposite signs, this will lead to a repelling force between the respective neurons. In this manner, connections between simultaneously firing neurons are strengthened while connections between counteracting neurons are weakened.

5.3 Boltzmann machine

A Boltzmann Machine (Hinton and Sejnowski, 1983) can be described as a Hopfield network with stochastic units, i.e. instead of assigning values deterministically we sample them from a distribution. The probabilities of the system states (associated with their respective energy levels) are modeled by a Boltzmann distribution. The stochasticity can be controlled by a *temperature* parameter T. A temperature of zero removes randomness from the system and makes it equivalent to a Hopfield Network, i.e. the Boltzmann machine is a generalization of the Hopfield Network.

Note: Boltzmann Distribution

Ludwig Eduard Boltzmann was a Austrian physicist who established the field of statistical mechanics in the 19th century. The Boltzmann distribution (often also referred to as Gibbs distribution) defines probabilities for a system to be in a certain state $i = 1, \ldots, M$. The states are associated with energy levels ε_i. Further, the distribution has a hyperparameter T representing the temperature of the system. The distribution is defined as

$$p(i) = \frac{\exp\left(-\frac{\varepsilon_i}{kT}\right)}{\sum_{j=1}^{M} \exp\left(-\frac{\varepsilon_j}{kT}\right)}, \tag{5.17}$$

where k is the Boltzmann constant, a factor that relates energy and temperature for particles in gas. The ratio of probabilities of the states i and j is

$$\frac{p(i)}{p(j)} = \exp\left(-\frac{\varepsilon_i - \varepsilon_j}{kT}\right). \tag{5.18}$$

From this we can compute the difference of their respective energy levels by

$$-(\varepsilon_i - \varepsilon_j) = kT \log p(i) - kT \log p(j). \tag{5.19}$$

Note that Equation (5.17) is mathematically equivalent to the *softmax* function.

Since the notions of energy and temperature in a Boltzmann Machine are abstract (i.e. need not correspond to energy and temperature in a physical sense), it is unnecessary to use the Boltzmann constant k in our notation to relate these terms and we may write T instead of kT.

We denote the neurons in the Boltzmann Machine by $\boldsymbol{x} \in \{0,1\}^D$ (and not $\{-1,+1\}^D$ as in Hopfield Networks). The state probabilities of a neuron are assumed to follow a Boltzmann distribution. There are only two possible states per neuron. Let ε_i denote the energy for $x_i = 0$ and ε_i' for $x_i = 1$. Then their probabilities simplify to

$$p(x_i = 0) = \frac{\exp\left(-\frac{\varepsilon_i}{T}\right)}{\exp\left(-\frac{\varepsilon_i}{T}\right) + \exp\left(-\frac{\varepsilon_i'}{T}\right)} = \frac{1}{1 + \exp\left(-\frac{\varepsilon_i' - \varepsilon_i}{T}\right)} = \sigma\left(\frac{\varepsilon_i' - \varepsilon_i}{T}\right) \tag{5.20}$$

and $p(x_i = 1) = 1 - p(x_i = 0) = \sigma((\varepsilon_i - \varepsilon_i')/T)$. If we write Equation (5.19) in this context

$$-(\varepsilon_i' - \varepsilon_i) = T \log p(x_i = 1) - T \log p(x_i = 0) \tag{5.21}$$

it expresses the difference in energy depending on whether unit x_i is off or on. From Equation (5.15) we infer that

$$\begin{aligned} -(\varepsilon_i' - \varepsilon_i) &= \left(\sum_{j=1}^{D} w_{ij} x_j - b_i\right)\left(x_i^{\text{on}} - x_i^{\text{off}}\right) \\ &= \sum_{j=1}^{D} w_{ij} x_j - b_i \,, \end{aligned} \tag{5.22}$$

where again $w_{ii} = 0$, i.e. it does not depend on the value x_i. The Boltzmann Machine runs by repeatedly choosing a neuron x_i, computing its input (5.22), and updating its value by sampling from the distribution (5.20). Running the network for long enough at temperature T, it will eventually converge to a Boltzmann distribution

$$p(\boldsymbol{x}) = \frac{\exp\left(-\frac{E(\boldsymbol{x})}{T}\right)}{\sum_{\boldsymbol{y} \in \{0,1\}^D} \exp\left(-\frac{E(\boldsymbol{y})}{T}\right)} \,, \tag{5.23}$$

where $\boldsymbol{x}$ is a binary state vector and $E(\boldsymbol{x})$ its associated energy. This is because the Boltzmann Machine is equivalent to a system of particles in contact with a heat bath at temperature T, which

will eventually reach its thermal equilibrium (Ackley et al., 1985). This is an important difference between Boltzmann Machines and Hopfield Networks. The former converge to a global energy minimum independent of the initial state while the latter will converge to a local minimum that is close to the initial state. Practical use of Boltzmann Machines? Autoencoders?

5.3.1 Learning in Boltzmann Machines

Given a set of state vectors $\{\boldsymbol{x}^{(n)}\}_{n=1}^{N}$, training a Boltzmann Machine is concerned with adjusting weights w_{ij} and biases b_i such that the Boltzmann distribution at thermal equilibrium assigns high probabilities to the states $\{\boldsymbol{x}^{(n)}\}_{n=1}^{N}$ in the training set.

5.3.2 Restricted Boltzmann Machines

5.3.3 Deep Boltzmann Machines

Chapter 6

Old Stuff from other Scripts

6.0.1 Sequence Processing with RNNs

The activation functions and network inputs are now time dependent and are computed from the activations of previous time step:

$$s_i(t) \;=\; \sum_{j=0}^{N} w_{ij}\; a_j(t-1) \tag{6.1}$$

$$a_i(t) \;=\; f(s_i(t))\;. \tag{6.2}$$

Further we need initial activations $a_i(0)$ which are the activations before the network sees the first input element.

Some of the units can be defined as input and some as output units.

For learning, for each input sequence $(\boldsymbol{x}(1), \ldots, \boldsymbol{x}(t), \ldots, \boldsymbol{x}(T))$, a sequence called target sequence $(\boldsymbol{y}(1), \ldots, \boldsymbol{y}(t), \ldots, \boldsymbol{y}(T))$ is given. We denote by $\{\boldsymbol{x}(t)\}$ the sequence $(\boldsymbol{x}(1), \ldots, \boldsymbol{x}(t))$ and by $\{\boldsymbol{y}(t)\}$ the sequence $(\boldsymbol{y}(1), \ldots, \boldsymbol{y}(t))$.

Similar to feedforward networks we define the empirical error as

$$\nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w}, \{\boldsymbol{X}(T)\}, \{\boldsymbol{Y}(T)\}) \;= \tag{6.3}$$
$$\frac{1}{l} \sum_{i=1}^{l} \sum_{t=1}^{T} \nabla_{\boldsymbol{w}} L\left(\boldsymbol{y}^i(t), \boldsymbol{g}\left(\{\boldsymbol{x}^i(t)\}; \boldsymbol{w}\right)\right)\;.$$

The next subsections discuss how recurrent networks can be learned.

6.0.2 Real-Time Recurrent Learning

For performing gradient descent we have to compute

$$\frac{\partial}{\partial w_{uv}}\, L\left(\boldsymbol{y}^i(t), \boldsymbol{g}\left(\{\boldsymbol{x}^i(t)\}; \boldsymbol{w}\right)\right) \tag{6.4}$$

for all w_{uv}.

Using the chain rule this can be expanded to

$$\frac{\partial}{\partial w_{uv}} L\left(\boldsymbol{y}^i(t), \boldsymbol{g}\left(\{\boldsymbol{x}^i(t)\}; \boldsymbol{w}\right)\right) = \sum_{k,k \text{ output unit}} \frac{\partial}{\partial a_k(t)} L\left(\boldsymbol{y}^i(t), \boldsymbol{g}\left(\{\boldsymbol{x}^i(t)\}; \boldsymbol{w}\right)\right) \frac{\partial a_k(t)}{\partial w_{uv}} . \tag{6.5}$$

For all units k we can compute

$$\frac{\partial a_k(t+1)}{\partial w_{uv}} = f'(s_k(t+1)) \left(\sum_l w_{kl} \frac{\partial a_l(t)}{\partial w_{uv}} + \delta_{ku} a_v(t) \right) , \tag{6.6}$$

where δ is the Kronecker delta with $\delta_{ku} = 1$ for $k = u$ and $\delta_{ku} = 0$ otherwise.

If the values $\frac{\partial a_k(t)}{\partial w_{uv}}$ are computed during a forward pass then this is called *real time recurrent learning* (RTRL) (Werbos, 1981; Robinson and Fallside, 1987; Williams and Zipser, 1989; Gherrity, 1989).

RTRL has complexity of $O(W^2)$ for a fully connected network, where $W = N\,(N - I) = O(N^2)$ (each unit is connected to each other unit except the input units, which do not have ingoing connections).

Note that RTRL is independent of the length of the sequence, therefore RTRL is *local in time*.

6.0.3 Back-Propagation Through Time

A recurrent network can be transformed into a feedforward network if for each time step a copy of all units is made. This procedure of *unfolding in time* is depicted in Fig. 2.1 for one step and the network unfolded over the whole time is depicted in Fig. 2.2.

After re-indexing the output by adding two time steps and re-indexing the hidden units by adding one time step we obtain the network from Fig. 6.1.

The network of Fig. 6.1 can now be trained by standard back-propagation. This method is called *back-propagation through time* (BPTT) (Williams and Zipser, 1992; Werbos, 1988; Pearlmutter, 1989; Werbos, 1990).

We have

$$L\left(\boldsymbol{y}^i, \boldsymbol{g}\left(\{\boldsymbol{x}^i\}; \boldsymbol{w}\right)\right) = \sum_{t=1}^{T} L\left(\boldsymbol{y}^i(t), \boldsymbol{g}\left(\{\boldsymbol{x}^i(t)\}; \boldsymbol{w}\right)\right) \tag{6.7}$$

for which we write for short

$$L = \sum_{t=1}^{T} L(t) . \tag{6.8}$$

The delta error is

$$\delta_j(t) = -\frac{\partial L}{\partial(s_j(t))} = -\frac{\partial\left(\sum_{\tau=t}^{T} L(\tau)\right)}{\partial(s_j(t))} \tag{6.9}$$

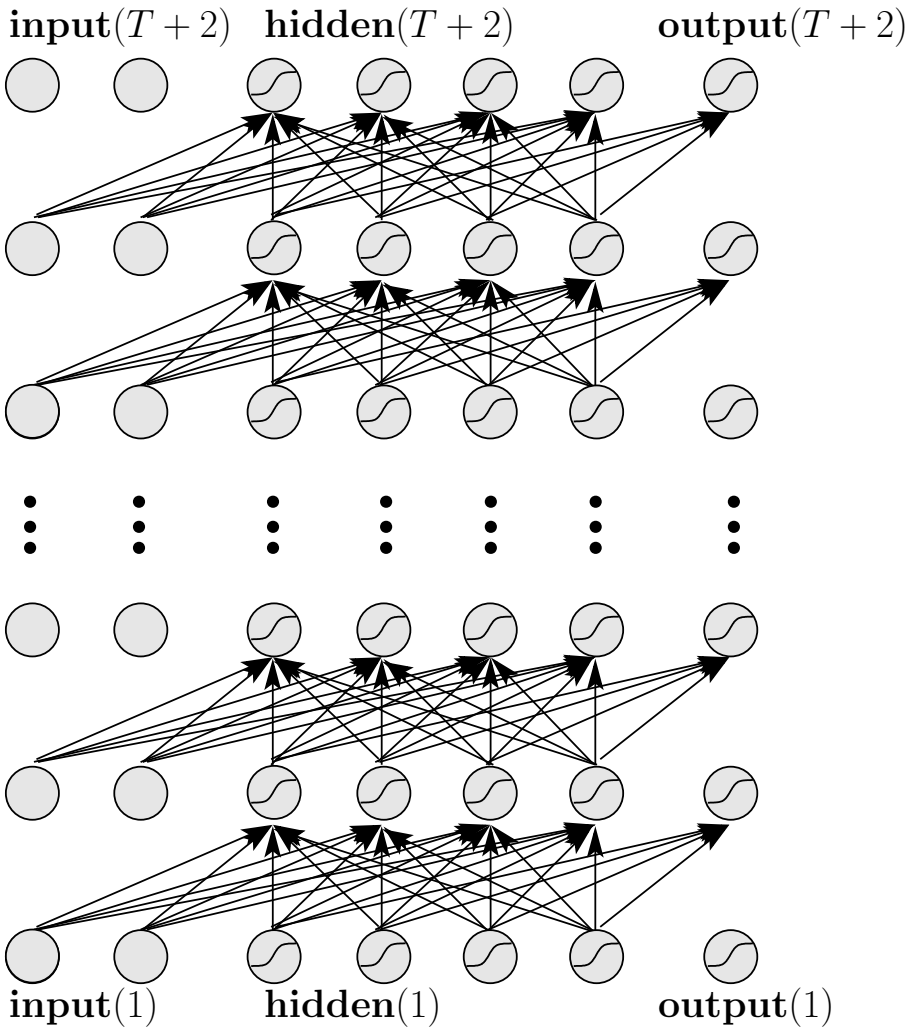


Figure 6.1: The recurrent network from Fig. 2.2 after re-indexing the hidden and output.

The back-propagation algorithm starts with

$$\delta_j(T) \;=\; f'(s_j(T))\,\frac{\partial L(T)}{\partial a_j(T)}\,, \tag{6.10}$$

where in above reformulation T has to be replaced by $(T+2)$.

For $t = T-1$ to $t = 1$:

$$\delta_j(t) \;=\; f'_j(s_j(t))\,\left(\frac{\partial L(t)}{\partial a_j(t)} \;+\; \sum_l w_{lj}\delta_l(t+1)\right)\,, \tag{6.11}$$

where $\frac{\partial L(t)}{\partial a_j(t)}$ accounts for the immediate error and the sum for the back-propagated error. Both types of error occur if output units have outgoing connections. In our architecture in Fig. 6.1 these errors are separated.

The derivative of the loss with respect to a weight in a layer t of the unfolded network in Fig. 6.1 is

$$-\,\frac{\partial L}{\partial w_{jl}(t)} \;=\; -\,\frac{\partial L}{\partial(s_j(t))}\,\frac{s_j(t)}{\partial w_{jl}(t)} \;=\; \delta_j(t)\;a_l(t-1)\,. \tag{6.12}$$

Because the corresponding weights in different layers are identical, the derivative of the loss with respect to a weight is

$$\frac{\partial L}{\partial w_{jl}} \;=\; \sum_{t=1}^{T}\frac{\partial L}{\partial w_{jl}(t)} \;=\; -\sum_{t=1}^{T}\left(\delta_j(t)\;a_l(t-1)\right)\,. \tag{6.13}$$

The on-line weight update is

$$w_{jl}^{\text{new}} \;=\; w_{jl}^{\text{old}} \;-\; \eta\,\frac{\partial L}{\partial w_{jl}}\,, \tag{6.14}$$

where η is the learning rate.

The complexity of BPTT is $O(T\,W)$. BPTT is *local in space* because its complexity per time step and weight is independent of the number of weights.

A special case of BPTT is truncated BPTT called BPTT(n), where only n steps is propagated back. For example $n = 10$ is sufficient because information further back in time cannot be learned to store and to process (see Subsection 6.0.5).

Therefore BPTT is in most cases faster than RTRL without loss of performance.

6.0.4 Other Approaches

We did not consider continuous time networks and did not consider attractor networks like the Hopfield model or Boltzmann machines.

These approaches are currently not relevant for bioinformatics. Continuous time networks may be useful for modeling e.g. for systems biology.

In the review (Pearlmutter, 1995) a nice overview over recurrent network approaches is given. The first approach were attractor networks for which gradient based methods were developed (Almeida, 1987; Pineda, 1987).

Other recurrent network architectures are

- Networks with context units which use special units, the context units, to store old activations. "Elman networks" (Elman, 1988) use as context units the old activations of hidden units. "Jordan networks" (Jordan, 1986) use as context units old activations of the output units.
- Special context units with time constants allow fast computation or to extract special information in the past. The "focused back-propagation" method (Mozer, 1989) and the method of (Gori et al., 1989) introduce delay factors for the context units and lead to fast learning methods.
- Other networks directly feed the output back for the next time step (Narendra and Parthasarathy, 1990) or NARX networks (Lin et al., 1996).
- Local recurrent networks use only local feedback loops to store information (Frasconi et al., 1992)
- Networks can be build on systems known from control theory like "Finite Impulse Response Filter" (FIR) networks (Wan, 1990) where at the ingoing weights a FIR filter is placed. The architecture in (Back and Tsoi, 1991) uses "Infinite Impulse Response Filter" (IIR) filter instead of FIR filter.
- Other networks use "Auto Regressive Moving Average" (ARMA) units.
- "Time Delay Neural Networks" (TDNNS) (Bodenhausen, 1990; Bodenhausen and Waibel, 1991) use connections with time delays, that means the signal is delayed as it gets transported over the connection. Therefore old inputs can be delayed until they are needed for processing.
- The "Gamma Memory" model (de Vries and Principe, 1991) is a combination of TDNN and time constant based methods.
- Recurrent network training can be based on the (extended) Kalman filter estimation (Matthews, 1990; Williams, 1992; Puskorius and Feldkamp, 1994).

A variant of RTRL is "Teacher Forcing-RTRL" if the output units also have outgoing connections. With "Teacher Forcing-RTRL" the activation of the output units is replaced by the target values.

RTRL and BPTT can be combined to a hybrid algorithm (Schmidhuber, 1992) which has complexity $O(W\ N)$. Here BPTT is made for N time steps having complexity $O(W\ N)$ whereafter a single RTRL update is made which as complexity $O(W^2/N) = O(W\ N)$.

6.0.5 Vanishing Gradient

The advantage of recurrent networks over feedforward networks with a window is that they can in principle use all information in the sequence which was so far processed.

However there is a problem in storing the relevant information over time. If recurrent networks are not able to store information over time then their great advantage over other approaches is lost.

Consider an error signal $\delta_u(t)$ which is computed at time t at unit u and which gets propagated back over q time steps to unit v. The effect of $\delta_u(t)$ on $\delta_v(t-q)$ can be computed as

$$\frac{\partial \delta_v(t-q)}{\partial \delta_u(t)} = \begin{cases} f'\left(s_v(t-1)\right)\ w_{uv} & q=1 \\ f'\left(s_v(t-q)\right)\ \sum_{l=I}^{N} \frac{\partial \delta_l(t-q+1)}{\partial \delta_u(t)} w_{lv} & q>1 \end{cases} . \tag{6.15}$$

If we set $l_q = v$ and $l_0 = u$ then we obtain

$$\frac{\partial \delta_v(t-q)}{\partial \delta_u(t)} = \sum_{l_1=I}^{N} \cdots \sum_{l_{q-1}=I}^{N} \prod_{r=1}^{q} f'\left(s_{l_r}(t-r)\right)\ w_{l_r l_{r-1}} . \tag{6.16}$$

Because of

$$\delta_v(t-q) = -\frac{\partial L}{\partial s_v(t-q)} = -\sum_l \frac{\partial L}{\partial s_l(t)} \frac{\partial s_l(t)}{\partial s_v(t-q)} = \tag{6.17}$$
$$\sum_l \delta_l(t)\ \frac{\partial s_l(t)}{\partial s_v(t-q)}$$

holds

$$\frac{\partial \delta_v(t-q)}{\partial \delta_u(t)} = \frac{\partial s_u(t)}{\partial s_v(t-q)} \tag{6.18}$$

true.

This leads to

$$\frac{\partial L(t)}{\partial w_{ij}(t-q)} = \tag{6.19}$$
$$\sum_{l:\text{output unit}} f'(s_u(t))\ \frac{\partial L(t)}{\partial a_j(t)}\ \frac{\partial s_l(t)}{\partial s_i(t-q)} \frac{\partial s_i(t-q)}{\partial w_{ij}(t-q)} =$$
$$\sum_{l:\text{output unit}} f'(s_u(t))\ \frac{\partial L(t)}{\partial a_j(t)}\ \frac{\partial \vartheta_i(t-q)}{\partial \vartheta_l(t)} a_j(t-q)$$

which shows that $\frac{\partial L(t)}{\partial w_{ij}(t-q)}$ is governed by the factor in eq. (6.16).

The eq. (6.16) contains N^{q-1} terms of the form

$$\prod_{r=1}^{q} f'\left(s_{l_r}(t-r)\right)\ w_{l_r l_{r-1}} . \tag{6.20}$$

If the multipliers

$$f'\left(s_{l_r}(t-r)\right)\ w_{l_r l_{r-1}}\ >\ 1\ , \tag{6.21}$$

the learning is instable because derivatives grow over time and the weight updates are too large.

If the multipliers

$$f'\left(s_{l_r}(t-r)\right)\ w_{l_r l_{r-1}}\ <\ 1\ , \tag{6.22}$$

then we have the case of *vanishing gradient* which means that $\frac{\partial L(t)}{\partial w_{ij}(t-q)}$ decreases with q exponentially to zero.

That means inputs which are far in the past do not influence the current loss and will not be used to improve the network. Therefore the network is not able to learn to store information in the past which can help to reduce the current loss.

6.0.6 Long Short-Term Memory

From previous considerations we know that to avoid both instable learning and the vanishing gradient, we have to enforce

$$f'\left(s_{l_r}(t-r)\right)\ w_{l_r l_{r-1}}\ =\ 1\ . \tag{6.23}$$

For simplicity we consider only one unit j with a self-recurrent connection w_{jj} and obtain

$$f'\left(s_j(t-r)\right)\ w_{jj}\ =\ 1\ . \tag{6.24}$$

Solving this differential equation gives:

$$f(x)\ =\ \frac{1}{w_{jj}}\ x\ . \tag{6.25}$$

Because f depends on the self-recurrent connection we can set $w_{jj} = 1$ to fix f and obtain

$$f(x)\ =\ x\ . \tag{6.26}$$

Fig. 6.2 shows the single unit which ensures that the vanishing gradient is avoided. Because this unit represents the identity it is immediately clear that information is stored over time through the architecture.

However it is not enough to avoid the vanishing gradient and therefore ensure keeping of information. A signal must be stored in the unit before it is kept. Fig. 6.3 shows the single unit with an additional input.

However a new problem appears: all information flowing over the input connection is stored. All information which is stored gets superimposed and the single signals cannot be accessed. More serious, not only relevant but also all irrelevant information is stored.

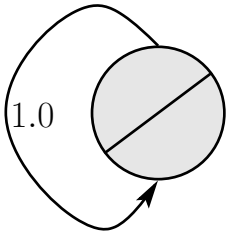


Figure 6.2: A single unit with self-recurrent connection which avoids the vanishing gradient.

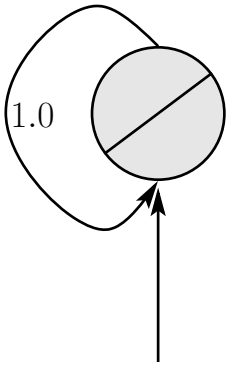


Figure 6.3: A single unit with self-recurrent connection which avoids the vanishing gradient and which has an input.

Only relevant information should be stored. In order to realize that an *input gate* a_{in} (basically a sigma-pi unit) is used. If we further assume that the network input s_{c} to the single unit is squashed then we obtain following dynamics:

$$a(t+1) \;=\; a(t) \;+\; a_{\mathrm{in}}(t)\, g\left(s_{\mathrm{c}}(t)\right) \;, \tag{6.27}$$

If a_{in} is a sigmoid unit active in $[0, b]$ then it can serve as a gating unit.

Now we assume that the output from the storing unit is also squashed by a function h. Further we can control the access to the stored information by an *output gate* a_{out} which is also a sigmoid unit active in $[0, b]$.

The memory access dynamics is

$$a_{\mathrm{c}}(t+1) \;=\; a_{\mathrm{out}}(t)\, h\left(a(t+1)\right) \;. \tag{6.28}$$

The whole dynamics is

$$a_{\mathrm{c}}(t+1) \;=\; a_{\mathrm{out}}(t)\, h\left(a(t) \;+\; a_{\mathrm{in}}(t)\, g\left(s_{\mathrm{c}}(t)\right)\right) \;. \tag{6.29}$$

The sub-architecture with the gate units a_{in} and a_{out} and the single unit a is called *memory cell* and depicted in Fig. 6.4.

The input to the storing unit is controlled by an "input gating" or attention unit (Fig. 6.4, unit marked "a_{in}") which blocks class-irrelevant information, so that only class-relevant information is stored in memory. The activation of attention units is bounded by 0 and b, i.e. the incoming

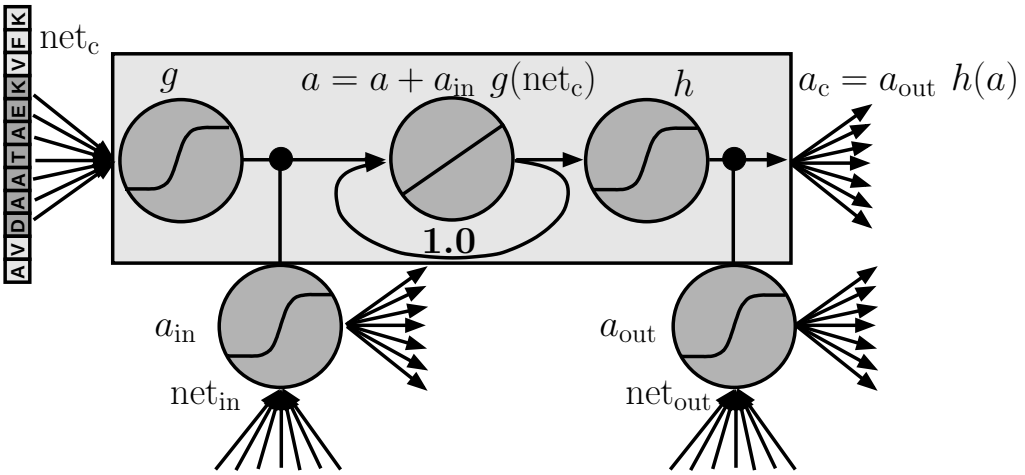


Figure 6.4: The LSTM memory cell. Arrows represent weighted connections of the neural network. Circles represent units (neurons), where the activation function (linear or sigmoid) is indicated in the circle.

information $s_{\mathrm{c}}(t)$ is squashed by a sigmoid function g. The output of the memory cell (Fig. 6.4, center) is bounded by the sigmoid function h (Fig. 6.4, unit labeled as "h"). Memory readout is controlled by an "output gate" (Fig. 6.4, unit labeled "a_{out}"). The cell's output a_{c} is then computed according to eq. (6.28) and eq. (6.29).

The recurrent network built from memory cells is called "Long Short-Term Memory" (LSTM, (Hochreiter and Schmidhuber, 1997c)). It is an RNN with a designed memory sub-architecture called "memory cell" to store information.

Memory cells can in principle be integrated into any neural network architecture. The LSTM recurrent network structure as depicted in Fig. 6.5.

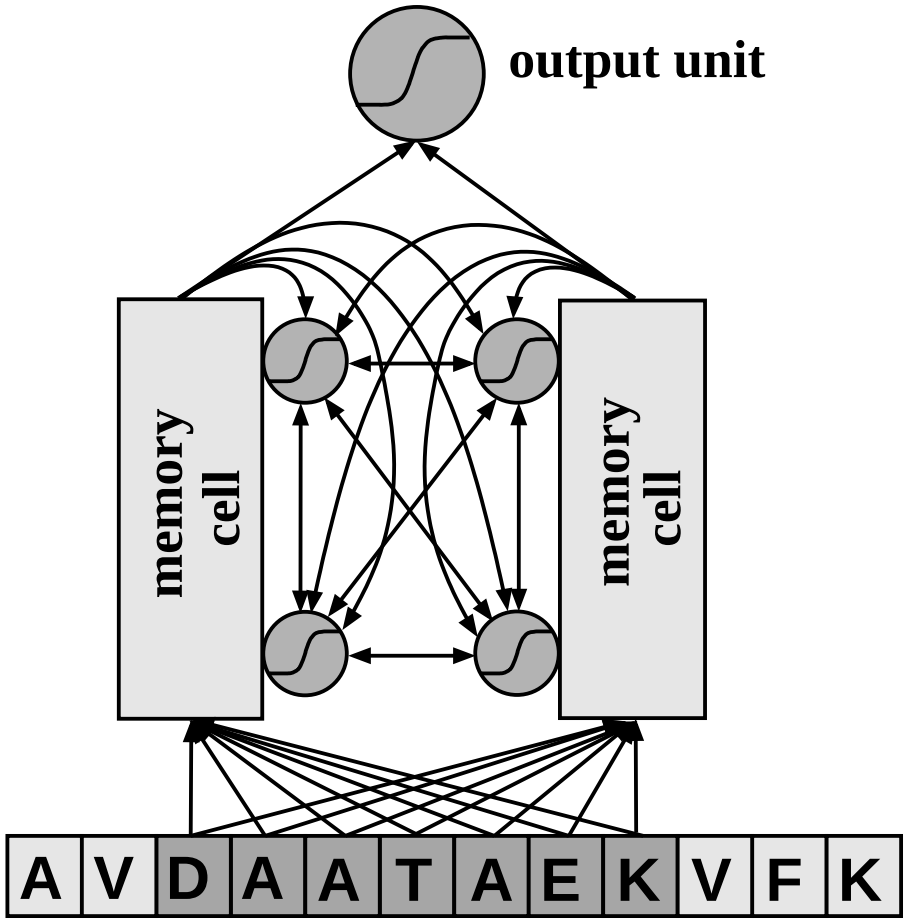


Figure 6.5: LSTM network with three layers: input layer (window of the amino acid sequence – shaded elements), hidden layer (with memory cells – see Fig. 6.4), and output layer. Arrows represent weighted connections; the hidden layer is fully connected. The subnetworks denoted by "memory cell" can store information and receive three kinds of inputs, which act as pattern detectors (input $\rightarrow$ hidden), as storage control (recurrent connections), and as retrieval control (recurrent connections). The stored information is combined at the output. The amino acid sequence is processed by scanning the sequence step by step from the beginning to the end.

Chapter 7

Hidden Markov Models

This is the first chapter devoted to unsupervised learning, especially to generative models. The most prominent generative model in bioinformatics is the hidden Markov model. It is well suited for analyzing protein or DNA sequences because of its discrete nature.

7.1 Hidden Markov Models in Bioinformatics

A hidden Markov model (HMM) is a generative approach for generating output sequences. The model is able to assign to each sequence a certain probability of being produced by the current model. The sequences of a class are used to build a model so that these sequences have high probability of being produced by the model. Thereafter the model can be utilized to search for sequences which also have high probability as being produced by the model. Therefore the new sequences with high probability are assumed to be similar to the sequences from which the model is build.

The HMM is able to model certain patterns in sequences and if those patterns are detected, the probability of the sequence is increased.

HMMs for gene prediction.

The DNA is scanned and exons and introns are identified from which the coding region of the gene can be obtain. Translating the coding regions of the gene gives a protein sequences. HMMs are a standard tool for identifying genes in a genome. GENSCAN (Burge and Karlin, 1997) and other HMM approaches to gene prediction (Kulp et al., 1996; Krogh, 1997; Krogh et al., 1994b) have a base-pair specificity between 50% and 80%.

Profile HMMs.

Profile HMMs (Krogh et al., 1994a) are used to store a multiple alignment in a hidden Markov model. An HMM is better suited for storing the alignment than a consensus string because it is a generative model. Especially new sequences can be be evaluated according to their likelihood of being produced by the model. Also the likelihood can be fine tuned after storing the alignment.

If HMMs are build from unaligned sequences, they often stick in local likelihood maxima. Approaches exist which try to avoid them, e.g. deterministic annealing (“Userguide” to HMMER version 1.8). Because of the poor results with unaligned sequences despite annealing approaches, in the new version of HMMER the HMMs are only initialized by alignment results.

The most common software packages for profile HMMs are HMMER (Eddy, 1998) (`http://hmmer.wustl.edu/`) and SAM (Krogh et al., 1994a) (`http://www.cse.ucsc.edu/compbio/sam.html`).

However HMMs have drawbacks as Sean Eddy writes (Eddy, 2004):

> "HMMs are reasonable models of linear sequence problems, but they don't deal well with correlations between residues or states, especially long-range correlations. HMMs assume that each residue depends only on one underlying state, and each state in the state path depends only on one previous state; otherwise, residues and states are independent of each other." ... "The state path of an HMM has no way of remembering what a distant state generated when a second state generates its residue."

Real valued protein characteristics like hydrophobic profiles cannot be represented by HMMs. HMMs cannot detect higher order correlations, cannot consider dependencies between regions in the sequence, cannot deal with correlations of elements within regions, cannot derive or process real valued protein characteristics, cannot take into account negative examples during model selection, and do not work sufficiently well for unaligned sequences.

Other HMMs Applications.

HMMs were used for remote homology detection (Park et al., 1998; Karplus et al., 1998, 1999) which is weak sequence homology (Sjölander et al., 1996).

HMMs were used for scoring (Barrett et al., 1997) and are combined with trees (Lio et al., 1999).

A whole data base is build on HMMs, namely the PFAM (protein family database) (Bateman et al., 2004, 2000). Here protein families are classified by HMMs. Software exists for large data sets or proteins like SMART (simple modular architecture research tool) (Schultz et al., 2000).

7.2 Hidden Markov Model Basics

A hidden Markov model (HMM) is a graph of connected hidden states $u \in \{1, \ldots, S\}$, where each state produces a probabilistic output.

Figure 7.1 shows a hidden Markov model with two state values 1 and 0 which are associated with "on" and "off". If the switch is "on" then it can remain on or go to the value "off". If the switch is "off" then it can remain off or go to the value "on". The state may be hidden in the sense that the position of the switch cannot be observed.

The model evolves over time t (in bioinformatics time is replaced by sequence position). At each step the process jumps from the current state into another state or remains in the current state. The evolving of the state variable u over time can be expressed by introducing the variable u_t for each time point t. At each time t the variable u_t has a certain value $u_t \in \{1, \ldots, S\}$. Fig. 7.2 shows a hidden Markov model where the state variable evolves over time.

It is possible to present all possible sequences of values of the hidden state like in Fig. 7.3. Each path in the figure from left to right is a possible sequence of state values. The probability of taking a certain value at a certain time (e.g. $u_5 = 3$) is the sum over all path' from the left to this value and time.

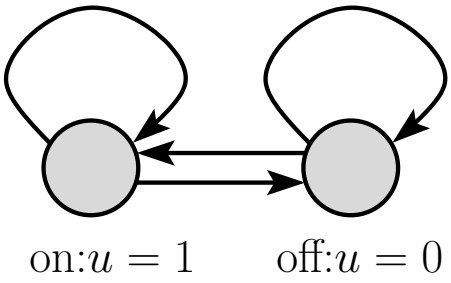


Figure 7.1: A simple hidden Markov model, where the state u can take on one of the two values 0 or 1. This model represents the switch for a light: it is either "on" or "off" and at every time point it can remain in its current state (reccurent connections) or go to the opposite state.

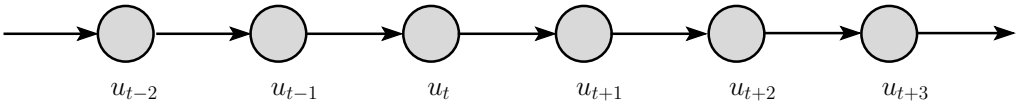


Figure 7.2: A simple hidden Markov model. The state u evolves over time and at each time t the state u takes on the value u_t.

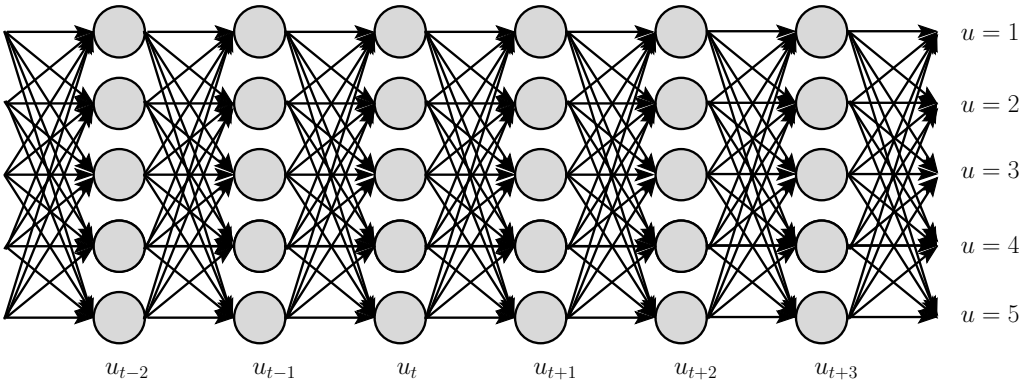


Figure 7.3: The hidden Markov model from Fig. 7.2 in more detail where also the state values $u = 1, \ldots, u = 5$ are given ($S = 5$). At each time t the state u_t takes on one of these values and if the state moves on the value may change. Each path from left to right has a certain probability and the probability of taking a certain value at a certain time is the sum over all paths from the left to this value and time.

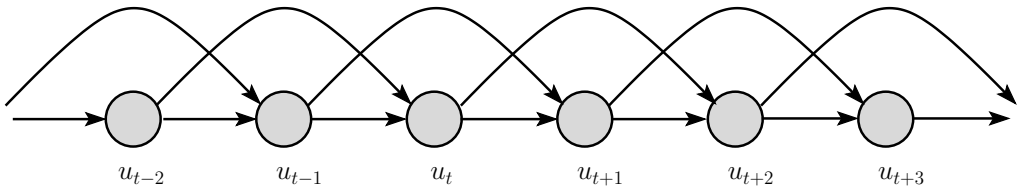


Figure 7.4: A second order hidden Markov model. The transition probability does not depend only on the current state value but also on the previous state value.

The hidden Markov model has transition probabilities $p(a \mid b)$, where $a, b \in \{1, \ldots, S\}$ and b is the current state and a the next state. Here the Markov assumption is that the next state only depends on the current state. Higher order hidden Markov models assume that the probability of going into the next state depends on the current state and previous states. For example in a second order Markov model the transition probability is $p(a \mid b, c)$, where $a, b, c \in \{1, \ldots, S\}$ and b is the current state, c the previous state, and a the next state. The second order Markov model is depicted in Fig. 7.4.

We will focus on a first order hidden Markov model, where the probability of going into a state depends only on the actual state.

At each time the state variable u takes on the value u_t and has a previous value given by u_{t-1}, therefore we observed the transition from u_{t-1} to u_t. This transition has probability of $p(u_t \mid u_{t-1})$,

Assume we have a certain start state probability $p_S(u_1)$ then the probability of observing the sequence $u^T = (u_1, u_2, u_3, \ldots, u_T)$ of length T is

$$p(u^T) \;=\; p_S(u_1) \prod_{t=2}^{T} p(u_t \mid u_{t-1}) \,. \tag{7.1}$$

For example a sequence may be $(u_1 = 3, u_2 = 5, u_3 = 2, \ldots, u_T = 4)$ that is the sequence $(3, 5, 2, \ldots, 4)$. Fig. 7.5 shows the hidden Markov model from Fig. 7.3 where now the transition probabilities are marked including the start state probability p_S.

Now we will consider Markov models which actually produce data, that means they are used as generative models. We assume that each state value has an emission probability $p_E(x_t \mid u_t)$ of emitting a certain output. Here u_t is a value of the state variable at time t (e.g. $u_t = 2$) and x_t is an element of the output alphabet of size P, for example $x_t \in \{A, T, C, G\}$. A specific emission probability may be $p_E(A \mid 2)$. Fig. 7.6 shows a hidden Markov model with output emission.

Fig. 7.7 shows a HMM where the output sequence is the Shine-Dalgarno pattern for ribosome binding regions.

Each output sequence has a certain probability of being produced by the current model. The joint probability of the output sequence $x^T = (x_1, x_2, x_3, \ldots, x_T)$ of length T and the hidden state value sequence $u^T = (u_1, u_2, u_3, \ldots, u_T)$ of length T is

$$p(u^T, x^T) \;=\; p_S(u_1) \prod_{t=2}^{T} p(u_t \mid u_{t-1}) \prod_{t=1}^{T} p_E(x_t \mid u_t) \,. \tag{7.2}$$

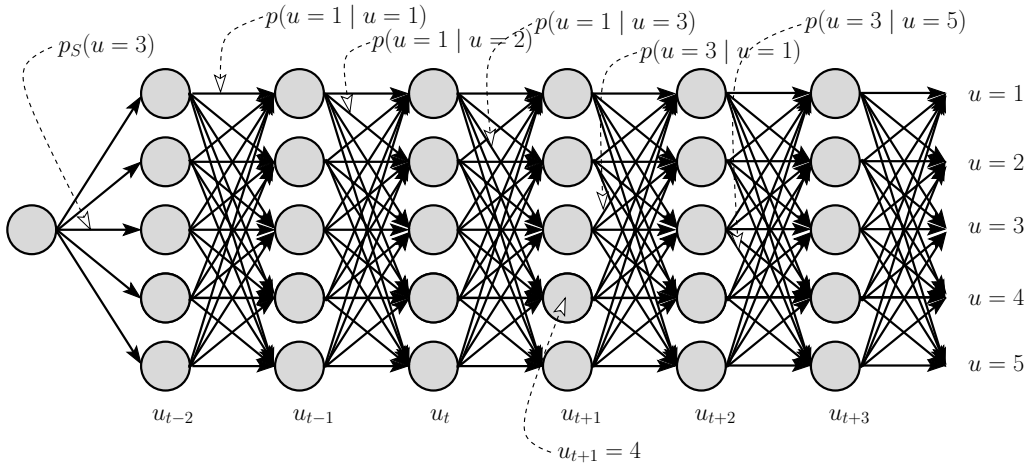


Figure 7.5: The hidden Markov model from Fig. 7.3 where now the transition probabilities are marked including the start state probability p_S. Also the state value $u_{t+1} = 4$ is marked.

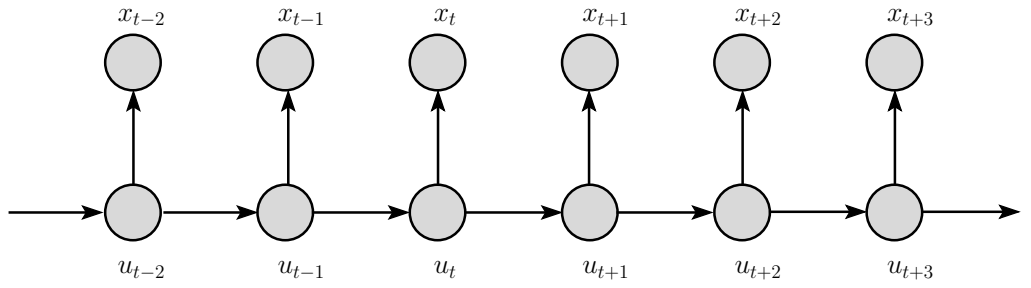


Figure 7.6: A simple hidden Markov model with output. At each time t the hidden state u_t has a certain probability of producing the output x_t.

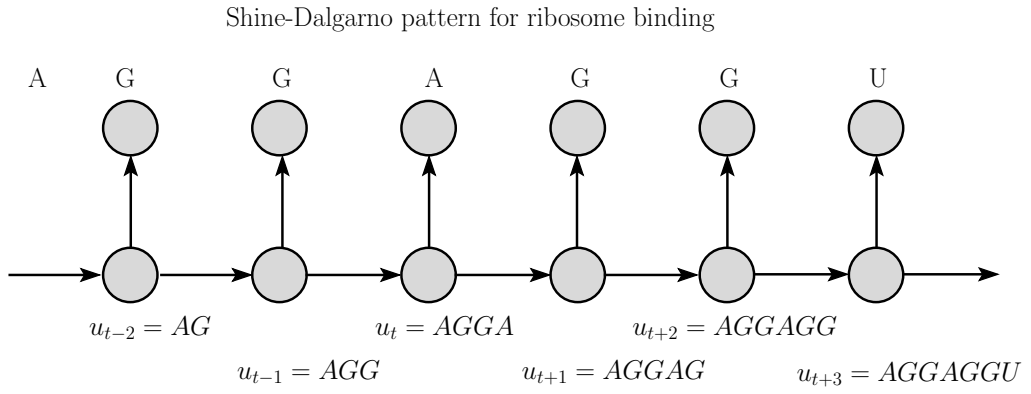


Figure 7.7: An HMM which supplies the Shine-Dalgarno pattern where the ribosome binds. Each state value is associated with a prefix sequence of the Shine-Dalgarno pattern. The state value for "no prefix" is omitted in the figure.

Through marginalization we obtain the probability of the output sequence x^T being produced by the HMM:

$$p(x^T) \;=\; \sum_{u^T} p(u^T, x^T) \;=\; \sum_{u^T} p_S(u_1) \prod_{t=2}^{T} p(u_t \mid u_{t-1}) \prod_{t=1}^{T} p_E(x_t \mid u_t)\,, \tag{7.3}$$

where $\sum_{u^T}$ denotes the sum over all possible sequences of length T of the values $\{1, \ldots, S\}$. The sum $\sum_{u^T}$ has S^T summands corresponding to different sequences (S values for u_1 multiplied by S values for u_2 etc.).

Fortunately, the (first order) Markov assumption allows to recursively compute above sum. We denote with $x^t = (x_1, x_2, x_3, \ldots, x_t)$ the prefix sequence of x^T of length t. We introduce the probability $p(x^t, u_t)$ of the model producing x^t and being in state u_t at the end.

$$\begin{aligned}
p(x^t, u_t) \;&=\; p(x_t \mid x^{t-1}, u_t)\, p(x^{t-1}, u_t) \;= \\
&p_E(x_t \mid u_t) \sum_{u_{t-1}} p(x^{t-1}, u_t, u_{t-1}) \;= \\
&p_E(x_t \mid u_t) \sum_{u_{t-1}} p(u_t \mid x^{t-1}, u_{t-1})\, p(x^{t-1}, u_{t-1}) \;= \\
&p_E(x_t \mid u_t) \sum_{u_{t-1}} p(u_t \mid u_{t-1})\, p(x^{t-1}, u_{t-1})\,,
\end{aligned} \tag{7.4}$$

where the Markov assumptions $p(x_t \mid x^{t-1}, u_t) \;=\; p_E(x_t \mid u_t)$ on the output emission and $p(u_t \mid x^{t-1}, u_{t-1}) \;=\; p(u_t \mid u_{t-1})$ on the transitions is used. Further marginalization $p(x^{t-1}, u_t) \;=\; \sum_{u_{t-1}} p(x^{t-1}, u_t, u_{t-1})$ and the definition of conditional probabilities $p(x^{t-1}, u_t, u_{t-1}) \;=\; p(u_t \mid x^{t-1}, u_{t-1})\, p(x^{t-1}, u_{t-1})$ were applied.

That means each recursion step needs only a sum over all u_{t-1} which is a sum over S values. However we have to do this for each value of u_t, therefore the recursion has complexity of $O(T\, S^2)$. The complexity can be reduced if transition probabilities are zero. The recursion starts with

$$p(x^1, u_1) \;=\; p_S(u_1)\, p_E(x_1 \mid u_1)\,. \tag{7.5}$$

The final probability of x^T can be computed as

$$p(x^T) \;=\; \sum_{u_T} p(x^T, u_T)\,. \tag{7.6}$$

This algorithm is called the "forward pass" or the "forward phase" and is used to compute the probability of x^T which is equal to the likelihood of x^T because we have discrete values. Alg. 7.1 shows the algorithm for the forward phase to compute the likelihood for one sequence for an HMM.

7.3 Expectation Maximization for HMM: Baum-Welch Algorithm

Now we focus on learning and parameter selection based on a training set.

Algorithm 7.1 HMM Forward Pass

Given: sequence $x^T = (x_1, x_2, x_3, \ldots, x_T)$, state values $u \in \{1, \ldots, S\}$, start probabilities $p_S(u_1)$, transition probabilities $p(u_t \mid u_{t-1})$, and emission probabilities $p_E(x_t \mid u_t)$; Output: likelihood $p(x^T)$ and $p(x^t, u_t)$

BEGIN initialization

$$p(x^1, u_1) \;=\; p_S(u_1)\, p_E(x_1 \mid u_1)$$

END initialization
BEGIN Recursion
for $(t = 2\,;\; t \leqslant T\,;\; t++)$ **do**
for $(a = 1\,;\; a \leqslant S\,;\; a++)$ **do**

$$p(x^t, u_t = a) \;=\; p_E(x_t \mid u_t = a) \sum_{u_{t-1}=1}^{S} p(u_t = a \mid u_{t-1})\, p(x^{t-1}, u_{t-1})$$

end for
end for
END Recursion
BEGIN Compute Likelihood

$$p(x^T) \;=\; \sum_{a=1}^{S} p(x^T, u_T = a)$$

END Compute Likelihood

The parameters of a hidden Markov model are the S start probabilities $p_S(u_1)$, the S^2 transitions probabilities $p(u_t \mid u_{t-1})$, and the $S\ P$ emission probabilities $p_E(x_t \mid u_t)$ (P is the number of output symbols).

If we have a set of training sequences $\{\boldsymbol{x}^i\}, 1 \leqslant i \leqslant l$, then the parameters can be optimized by maximizing the likelihood.

Instead of gradient based methods we deduce an Expectation Maximization algorithm. In the lectures "Machine Learning: Unsupervised Techniques" and "Theoretical Concepts of Machine Learning" we defined

$$\mathcal{F}(Q, \boldsymbol{w}) \ = \ \int_U Q(\boldsymbol{u} \mid \boldsymbol{x}) \ \ln p(\boldsymbol{x}, \boldsymbol{u}; \boldsymbol{w}) \ d\boldsymbol{u} \ - \tag{7.7}$$
$$\int_U Q(\boldsymbol{u} \mid \boldsymbol{x}) \ \ln Q(\boldsymbol{u} \mid \boldsymbol{x}) \ d\boldsymbol{u} \ ,$$

where $Q(\boldsymbol{u} \mid \boldsymbol{x})$ is an estimation for $p(\boldsymbol{u} \mid \boldsymbol{x}; \boldsymbol{w})$.

We have to adapt this formulation to discrete HMMs. For HMMs $\boldsymbol{u}$ is the sequence of hidden states, $\boldsymbol{x}$ the sequence of output states and $\boldsymbol{w}$ summarizes all probability parameters (start, transition, and emission) in the model. The integral $\int_U d\boldsymbol{u}$ can be replaced by a sum.

The estimation for the state sequence can be written as

$$Q(\boldsymbol{u} \mid \boldsymbol{x}) \ = \ p(u_1 = a_1, u_2 = a_2, \ldots, u_T = a_T \mid x^T; \boldsymbol{w}) \ . \tag{7.8}$$

We obtain

$$\mathcal{F}(Q, \boldsymbol{w}) \ = \ \sum_{a_1=1}^{S} \ldots \sum_{a_T=1}^{S} \tag{7.9}$$
$$p(u_1 = a_1, u_2 = a_2, \ldots, u_T = a_T \mid x^T; \boldsymbol{w}) \ \ln p(x^T, u^T; \boldsymbol{w}) \ -$$
$$\sum_{a_1=1}^{S} \ldots \sum_{a_T=1}^{S} p(u_1 = a_1, u_2 = a_2, \ldots, u_T = a_T \mid x^T; \boldsymbol{w})$$
$$\ln p(u_1 = a_1, u_2 = a_2, \ldots, u_T = a_T \mid x^T; \boldsymbol{w}) \ = \tag{7.10}$$
$$\sum_{a_1=1}^{S} \ldots \sum_{a_T=1}^{S} p(u_1 = a_1, u_2 = a_2, \ldots, u_T = a_T \mid x^T; \boldsymbol{w}) \ \ln p(x^T, u^T; \boldsymbol{w})$$
$$-c \ ,$$

where c is a constant independent of $\boldsymbol{w}$.

We have

$$\ln p(u^T, x^T; \boldsymbol{w}) \ = \ \ln p_S(u_1) \ + \ \sum_{t=2}^{T} \ln p(u_t \mid u_{t-1}) \ + \ \sum_{t=1}^{T} \ln p_E(x_t \mid u_t) \ . \tag{7.11}$$

Because most variables a_t can be summed out we obtain:

$$\begin{aligned}\mathcal{F}(Q, \boldsymbol{w}) \;=\; & \sum_{a=1}^{S} p(u_1 = a \mid x^T; \boldsymbol{w}) \;\ln p_S(u_1 = a) \;+ \\ & \sum_{t=1}^{T} \sum_{a=1}^{S} p(u_t = a \mid x^T; \boldsymbol{w}) \;\ln p_E(x_t \mid u_t = a) \;+ \\ & \sum_{t=2}^{T} \sum_{a=1}^{S} \sum_{b=1}^{S} p(u_t = a, u_{t-1} = b \mid x^T; \boldsymbol{w}) \;\ln p(u_t = a \mid u_{t-1} = b) \;- c\,.\end{aligned} \tag{7.12}$$

Note that the parameters $\boldsymbol{w}$ are the start probabilities $p_S(a)$, the emission probabilities $p_E(x \mid a)$, and the transition probabilities $p(a \mid b)$. We have as constraints $\sum_a p_S(a) \;=\; 1$, $\sum_x p_E(x \mid a) \;=\; 1$, and $\sum_a p(a \mid b) \;=\; 1$.

Consider the optimization problem

$$\begin{aligned}\min_{\boldsymbol{w}} \qquad & \sum_t \sum_a c_{ta} \;\ln w_a \\ \text{s.t.} \qquad & \sum_a w_a \;=\; 1\,.\end{aligned} \tag{7.13}$$

The Lagrangian is

$$L \;=\; \sum_t \sum_a c_{ta} \;\ln w_a \;-\; \lambda \left(\sum_a w_a \;-\; 1 \right)\,. \tag{7.14}$$

Optimality requires

$$\frac{\partial L}{\partial w_a} \;=\; \sum_t c_{ta} \;\frac{1}{w_a} \;-\; \lambda \;=\; 0 \tag{7.15}$$

therefore

$$\sum_t c_{ta} \;-\; \lambda \, w_a \;=\; 0 \tag{7.16}$$

and summing over a gives

$$\sum_a \sum_t c_{ta} \;=\; \lambda\,. \tag{7.17}$$

We obtain

$$w_a \;=\; \frac{\sum_t c_{ta}}{\sum_a \sum_t c_{ta}}\,. \tag{7.18}$$

The constraint maximization step (M-step) is therefore

$$p_S(a) \;=\; \frac{p(u_1 = a \mid x^T; \boldsymbol{w})}{\sum_{a'} p(u_1 = a' \mid x^T; \boldsymbol{w})} \tag{7.19}$$

$$p_E(x \mid a) \;=\; \frac{\sum_{t=1}^{T} \delta_{x_t = x} \; p(u_t = a \mid x^T; \boldsymbol{w})}{\sum_y \sum_{t=1}^{T} \delta_{x_t = y} \; p(u_t = a \mid x^T; \boldsymbol{w})} \tag{7.20}$$

$$p(a \mid b) \;=\; \frac{\sum_{t=2}^{T} p(u_t = a, u_{t-1} = b \mid x^T; \boldsymbol{w})}{\sum_{a'} \sum_{t=2}^{T} p(u_t = a', u_{t-1} = b \mid x^T; \boldsymbol{w})} \tag{7.21}$$

which is

$$p_S(a) \;=\; p(u_1 = a \mid x^T; \boldsymbol{w}) \tag{7.22}$$

$$p_E(x \mid a) \;=\; \frac{\sum_{t=1}^{T} \delta_{x_t = x}\, p(u_t = a \mid x^T; \boldsymbol{w})}{\sum_{t=1}^{T} p(u_t = a \mid x^T; \boldsymbol{w})} \tag{7.23}$$

$$p(a \mid b) \;=\; \frac{\sum_{t=2}^{T} p(u_t = a, u_{t-1} = b \mid x^T; \boldsymbol{w})}{\sum_{t=2}^{T} p(u_{t-1} = b \mid x^T; \boldsymbol{w})} \;. \tag{7.24}$$

We now consider the estimation step (E-step) in order to estimate $p(u_t = a \mid x^T; \boldsymbol{w})$ and $p(u_t = a, u_{t-1} = b \mid x^T; \boldsymbol{w})$. First we have to introduce the suffix sequence $x^{t \leftarrow T} = (x_t, x_{t+1}, \ldots, x_T)$. of length $T - t + 1$.

We use the probability $p(x^{t+1 \leftarrow T} \mid u_t = a)$ of the suffix sequence $x^{t+1 \leftarrow T} = (x_{t+1}, \ldots, x_T)$ being produced by the model if starting from $u_t = a$.

Now we can formulate an expression for $p(u_t = a \mid x^T; \boldsymbol{w})$

$$p(u_t = a \mid x^T; \boldsymbol{w}) \;=\; \frac{p(u_t = a, x^T; \boldsymbol{w})}{p(x^T)} \;= \tag{7.25}$$
$$\frac{p(x^t, u_t = a; \boldsymbol{w})\; p(x^{t+1 \leftarrow T} \mid u_t = a)}{p(x^T)} \;,$$

where the first "=" is the definition of conditional probability and the second "=" says that all paths of hidden values which have at time t the value a can be separated into a prefix path from start to time t ending in the value a and a suffix path starting at time t in a.

Similar we can formulate an expression for $p(u_t = a, u_{t-1} = b \mid x^T; \boldsymbol{w})$

$$p(u_t = a, u_{t-1} = b \mid x^T; \boldsymbol{w}) \;=\; \frac{p(u_t = a, u_{t-1} = b, x^T; \boldsymbol{w})}{p(x^T)} \;= \tag{7.26}$$
$$p(x^{t-1}, u_{t-1} = b; \boldsymbol{w})\; p(u_t = a \mid u_{t-1} = b)\; p_E(x_t \mid u_t = a)$$
$$p(x^{t+1 \leftarrow T} \mid u_t = a) \;/\; p(x^T) \;,$$

where again the first "=" is the conditional probability and the second "=" says all paths which are at time t in state value a and in time $(t-1)$ in state value b can be separated in a prefix path from start to time $(t-1)$ ending in b, a suffix path starting from t in value a to the end, the transition $b \leftarrow a$ with probability $p(u_t = a \mid u_{t-1} = b)$ and the emission of x_t given by $p_E(x_t \mid u_t = a)$.

Note that

$$p(u_t = a \mid x^T; \boldsymbol{w}) \;=\; \sum_{b=1}^{S} p(u_t = a, u_{t-1} = b \mid x^T; \boldsymbol{w}) \;. \tag{7.27}$$

Similar to eq. (7.4) we can derive a backward recursion for computing $p(x^{t+1 \leftarrow T} \mid u_t = a)$ by using the Markov assumptions:

$$p(x^{t+1 \leftarrow T} \mid u_t = a) \;= \tag{7.28}$$
$$\sum_{b=1}^{S} p_E(x_{t+1} \mid u_{t+1} = b)\; p(u_{t+1} = b \mid u_t = a)\; p(x^{t+2 \leftarrow T} \mid u_{t+1} = b) \;.$$

The starting conditions are

$$p(x^{T\leftarrow T} \mid u_{T-1} = a) \;=\; \sum_{b=1}^{S} p_E(x_T \mid u_T = b)\, p(u_T = b \mid u_{T-1} = a) \tag{7.29}$$

or, alternatively

$$\forall_a : \;\; p(x^{T+1\leftarrow T} \mid u_T = a) \;=\; 1 \tag{7.30}$$

In Alg. 7.2 an algorithm for the backward procedure for HMMs is given.

Algorithm 7.2 HMM Backward Pass

Given: sequence $x^T = (x_1, x_2, x_3, \ldots, x_T)$, state values $u \in \{1, \ldots, S\}$, start probabilities $p_S(u_1)$, transition probabilities $p(u_t \mid u_{t-1})$, and emission probabilities $p_E(x_t \mid u_t)$; Output: likelihood $p(x^T)$ and $p(x^{t+1\leftarrow T} \mid u_t = a)$

BEGIN initialization

$$\forall_a : \;\; p(x^{T+1\leftarrow T} \mid u_T = a) \;=\; 1$$

END initialization
BEGIN Recursion
for $(t = T-1 \;;\; t \geqslant 1 \;;\; t--)$ **do**
for $(a = 1 \;;\; a \leqslant S \;;\; a++)$ **do**

$$p(x^{t+1\leftarrow T} \mid u_t = a) \;= \\ \sum_{b=1}^{S} p_E(x_{t+1} \mid u_{t+1} = b)\, p(u_{t+1} = b \mid u_t = a)\, p(x^{t+2\leftarrow T} \mid u_{t+1} = b)\;.$$

end for
end for
END Recursion
BEGIN Compute Likelihood

$$p(x^T) \;=\; \sum_{a=1}^{S} p_S(u_1 = a)\; (p(x^{1\leftarrow T} \mid u_1 = a)$$

END Compute Likelihood

The EM algorithm for HMMs is given in Alg. 7.3 which is based on the forward procedure Alg. 7.1 and the backward procedure Alg. 7.2.

Algorithm 7.3 HMM EM Algorithm

Given: l training sequences $(x^T)^i = (x_1^i, x_2^i, x_3^i, \ldots, x_T^i)$ for $1 \leqslant i \leqslant l$, state values $u \in \{1, \ldots, S\}$, start probabilities $p_S(u_1)$, transition probabilities $p(u_t \mid u_{t-1})$, and emission probabilities $p_E(x \mid u)$; Output: updated values of $p_S(u)$, $p_E(x \mid u)$, and $p(u_t \mid u_{t-1})$

BEGIN initialization

initialize start probabilities $p_S(u_1)$, transition probabilities $p(u_t \mid u_{t-1})$, and emission probabilities $p_E(x \mid u)$; Output: updated values of $p_S(u)$, $p_E(x \mid u)$, and $p(u_t \mid u_{t-1})$

END initialization

Stop=false

while Stop=false **do**

for $(i = 1 \,;\, i \geqslant l \,;\, i++)$ **do**

Forward Pass

forward pass for $(x^T)^i$ according to Alg. 7.1

Backward Pass

backward pass for $(x^T)^i$ according to Alg. 7.2

E-Step

for $(a = 1 \,;\, a \leqslant S \,;\, a++)$ **do**

for $(b = 1 \,;\, b \leqslant S \,;\, b++)$ **do**

$$p(u_t = a, u_{t-1} = b \mid (x^T)^i; \boldsymbol{w}) = p((x^{t-1})^i, u_{t-1} = b; \boldsymbol{w})\, p(u_t = a \mid u_{t-1} = b)\, p_E(x_t^i \mid u_t = a)\, p((x^{t+1 \leftarrow T})^i \mid u_t = a) \,/\, p((x^T)^i)$$

end for

end for

for $(a = 1 \,;\, a \leqslant S \,;\, a++)$ **do**

$$p(u_t = a \mid (x^T)^i; \boldsymbol{w}) = \sum_{b=1}^{S} p(u_t = a, u_{t-1} = b \mid (x^T)^i; \boldsymbol{w})$$

end for

M-Step

for $(a = 1 \,;\, a \leqslant S \,;\, a++)$ **do**

$$p_S(a) = p(u_1 = a \mid (x^T)^i; \boldsymbol{w})$$

end for

for $(a = 1 \,;\, a \leqslant S \,;\, a++)$ **do**

for $(x = 1 \,;\, x \leqslant P \,;\, x++)$ **do**

$$p_E(x \mid a) = \frac{\sum_{t=1}^{T} \delta_{x_t^i = x}\, p(u_t = a \mid (x^T)^i; \boldsymbol{w})}{\sum_{t=1}^{T} p(u_t = a \mid (x^T)^i; \boldsymbol{w})}$$

end for

end for

for $(a = 1 \,;\, a \leqslant S \,;\, a++)$ **do**

for $(b = 1 \,;\, b \leqslant S \,;\, b++)$ **do**

$$p(a \mid b) = \frac{\sum_{t=2}^{T} p(u_t = a, u_{t-1} = b \mid (x^T)^i; \boldsymbol{w})}{\sum_{t=2}^{T} p(u_{t-1} = b \mid (x^T)^i; \boldsymbol{w})}$$

end for

end for

end for

if stop criterion fulfilled **then**

Stop=true

end if

end while

7.4 Viterby Algorithm

In the forward (and also backward) pass we computed $p(x^T)$, the probability of producing x^T by the model, that is the likelihood of x^T. The likelihood of x^T is an integral – more exactly a sum – over all probabilities of possible sequences of hidden states multiplied by the probability that the hidden sequence emits x^T.

In many cases a specific hidden sequence $(u^T)^* = (u_1^*, u_2^*, u_3^*, \ldots, u_T^*)$ and its probability of emitting x^T dominates the above sum. More formally

$$(u^T)^* \;=\; \arg\max_{u^T} p(u^T \mid x^T) \;=\; \arg\max_{u^T} p(u^T, x^T)\;. \tag{7.31}$$

$(u^T)^*$ is of interest if the hidden states have a semantic meaning, then one want to extract $(u^T)^*$.

In bioinformatics the extraction of $(u^T)^*$ is important to make an alignment of a sequence with a multiple alignment stored in an HMM.

Because $(u^T)^*$ can be viewed as an alignment, it is not surprising that it can be obtained through dynamic programming. The dynamic programming algorithm has to find a path in Fig. 7.5 from left to right. Towards this end the state values at a certain time which are the circles in Fig. 7.5 are represented by a matrix $\boldsymbol{V}$. $V_{t,a}$ contains the maximal probability of a sequence of length t ending in state value a:

$$V_{t,a} \;=\; \max_{u^{t-1}} p(x^t, u^{t-1}, u_t = a)\;. \tag{7.32}$$

The Markov conditions allow now to formulate $V_{t,a}$ recursively:

$$V_{t,a} \;=\; p_E(x_t \mid u_t = a)\; \max_b p(u_t = a \mid u_{t-1} = b)\; V_{t-1,b} \tag{7.33}$$

with initialization

$$V_{1,a} \;=\; p_S(a) p_E(x_1 \mid u_1 = a) \tag{7.34}$$

and the result

$$\max_{u^T} p(u^T, x^T) \;=\; \max_a V_{T,a}\;. \tag{7.35}$$

The best sequence of hidden states can be found by back-tracing using

$$b(t,a) \;=\; \arg\max_b p(u_t = a \mid u_{t-1} = b)\; V_{t-1,b} \tag{7.36}$$

The complexity of the Viterby algorithm is $O(T\ S^2)$ because all $S\ T$ values $V_{t,a}$ must be computed and for computing them, the maximum over S terms must be determined.

The Viterby algorithm can be used to iteratively improve a multiple alignment:

1 initialize the HMM

2 align all sequences to the HMM via the Viterby algorithm

3 make frequency counts per column and compute the transition probabilities to update the HMM

4 if not converged go to step 2

Algorithm 7.4 HMM Viterby

Given: sequence $x^T = (x_1, x_2, x_3, \ldots, x_T)$, state values $u \in \{1, \ldots, S\}$, start probabilities $p_S(u_1)$, transition probabilities $p(u_t \mid u_{t-1})$, and emission probabilities $p_E(x_t \mid u_t)$; Output: most likely sequence of hidden state values $(u^T)^*$ and its probability $p\left(x^T, (u^T)^*\right)$

BEGIN initialization

$$V_{1,a} = p_S(a) p_E(x_1 \mid u_1 = a)$$

END initialization
BEGIN Recursion
for $(t = 2\ ;\ t \leqslant T\ ;\ t++)$ **do**
for $(a = 1\ ;\ a \leqslant S\ ;\ a++)$ **do**

$$V_{t,a} = p_E(x_t \mid u_t = a) \max_b p(u_t = a \mid u_{t-1} = b)\, V_{t-1,b}$$

$$b(t, a) = \arg\max_b p(u_t = a \mid u_{t-1} = b)\, V_{t-1,b}$$

end for
end for
END Recursion
BEGIN Compute Probability

$$p\left(x^T, (u^T)^*\right) = \max_{a=1}^{S} V(T, a)$$

END Compute Probability
BEGIN Back-tracing

$$s = \arg\max_{a=1}^{S} V(T, a)$$

print s

for $(t = T\ ;\ t \geqslant 2\ ;\ t--)$ **do**

$$s = b(t, s)$$

print s

end for
END Back-tracing

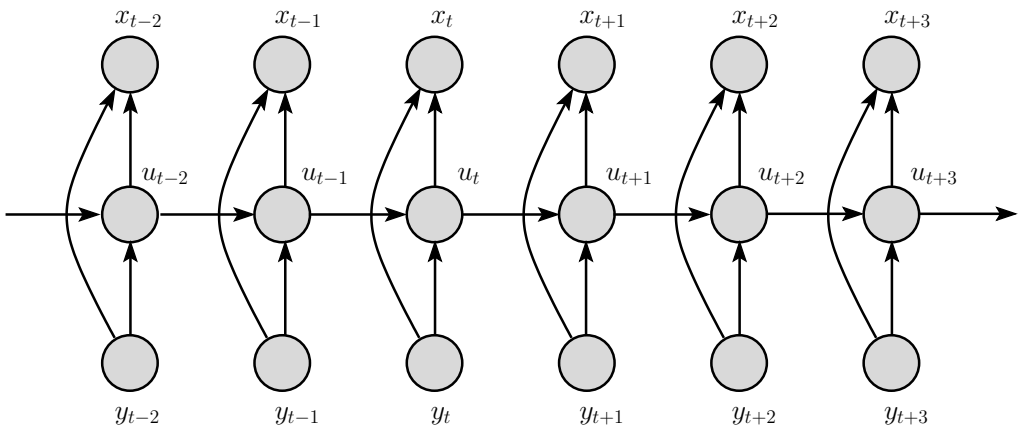


Figure 7.8: An input output HMM (IOHMM) where the output sequence $x^T = (x_1, x_2, x_3, \ldots, x_T)$ is conditioned on the input sequence $y^T = (y_1, y_2, y_3, \ldots, y_T)$.

7.5 Input Output Hidden Markov Models

Input Output Hidden Markov Models (IOHMMs) generate an output sequence $x^T = (x_1, x_2, x_3, \ldots, x_T)$ of length T conditioned on an input sequence $y^T = (y_1, y_2, y_3, \ldots, y_T)$ of length T.

The difference between standard HMMs and input output HMMs is that the probabilities are conditioned on the input. Start probabilities are $p_S(u_1 \mid y_1)$, the transition probabilities $p(u_t \mid y_t, u_{t-1})$, and the emission probabilities $p_E(x_t \mid y_t, u_t)$.

Using IOMMs also negative examples can be used by setting for all y_t either a don't care or a fixed value and setting $y_T = 1$ for the positive class and $y_T = -1$ for the negative class. Whether a model for the negative class can be built is not clear but at least a subclass of the negative class which is very similar to the positive class can be better discriminated.

The number of parameters increase proportional to the number of input symbols, which may make it more difficult to estimate the probabilities if not enough data is available.

Learning via the likelihood is as with the standard HMM with the probabilities additionally conditioned on the input.

7.6 Factorial Hidden Markov Models

The HMM architecture Fig. 7.6 is extended to Fig. 7.9 where the hidden state is divided into more components u_i (three in the figure).

The transition probability of u_i is conditioned on all u_k with $k \leqslant i$ and the emission probability depends on all hidden states. In the HMM architecture in Fig. 7.9 u_1 evolves very slowly, u_2 evolves faster, and u_3 evolves fastest of all hidden variables. Fast evolving variables do not influence slow evolving ones but slow evolving variables influence fast evolving variables.

If the factorial HMM has h hidden state variables u_i and each one of them can take on S values then the emission probability distribution consists of $P\ S^h$ emission probabilities. Therefore

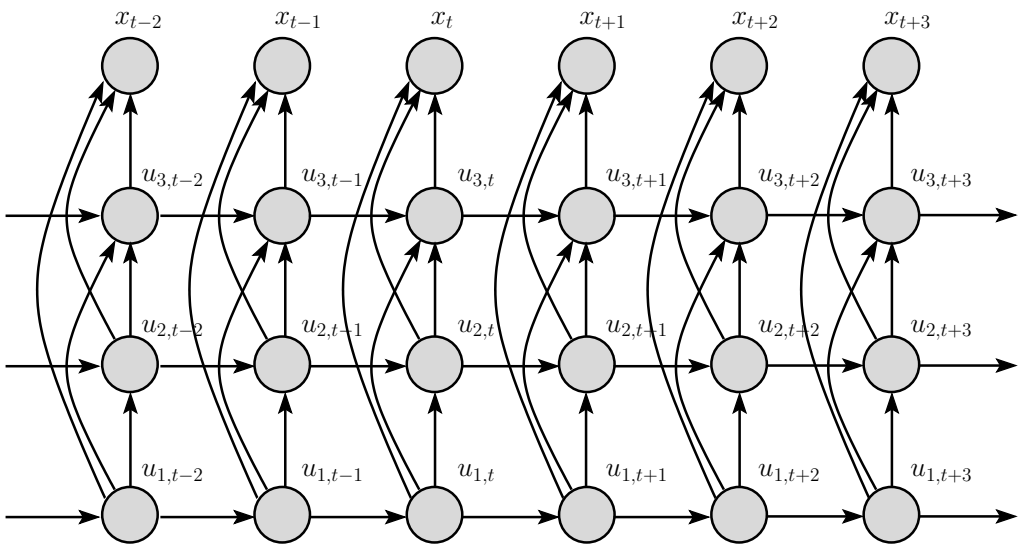


Figure 7.9: A factorial HMM with three hidden state variables u_1, u_2, and u_3. The transition probability of u_i is conditioned on all u_k with $k \leqslant i$ and the emission probability depends on all hidden states.

learning factorial HMMs is computational expensive. However approximative methods have been developed to speed up learning (Ghahramani and Jordan, 1996, 1997).

7.7 Memory Input Output Factorial Hidden Markov Models

Remember that we quoted Sean Eddy (Eddy, 2004):

> "HMMs are reasonable models of linear sequence problems, but they don't deal well with correlations between residues or states, especially long-range correlations. HMMs assume that each residue depends only on one underlying state, and each state in the state path depends only on one previous state; otherwise, residues and states are independent of each other." ... "The state path of an HMM has no way of remembering what a distant state generated when a second state generates its residue."

The only way a HMM can store information over time is to go into a certain state value and don't change it any more. The state is fixed and the event which led the HMM enter the fixed state is memorized. Instead of a state a set of states can be entered from which the escape probability is zero.

To realize a state with a non-escaping value which can memorize past events is

$$p(u_t = a \mid u_{t-1} = a) \;=\; 1 \,.$$

That means if the state takes on the value a then the state will not take any other value.

In principle the storage of past events can be learned but the likelihood of storing decreases exponentially with the time of storage. Therefore learning to store is practically impossible because

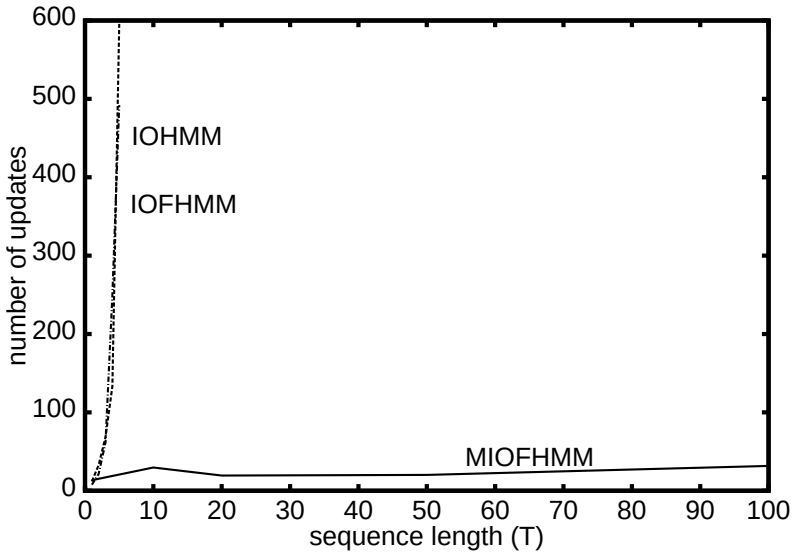


Figure 7.10: Number of updates required to learn to remember an input element until sequence end for three models: input output HMM (IOHMM), input output factorial HMM (IOFHMM), and "Memory-based Input-Output Factorial HMM" (MIOFHMM) as a function of the sequence length T.

these small likelihood differences are tiny in comparison to local minima resulting from certain input / output patterns or input / output distributions.

Therefore memory is enforced by setting $p(u_t = a \mid u_{t-1} = a) = 1$ and not allowing this probability to change.

However after the storage process (taking on the value a) the model is fixed and neither future systems dynamics nor other events to memorize can be dealt with.

To overcome this problem a factorial HMM can be used where some of the hidden state variables can store information and others extract the dynamics of the system to model.

Storing events is especially suited for input output HMMs where input events can be stored.

An architecture with memory state variable and using the input output architecture is the "Memory-based Input-Output Factorial HMM" (MIOFHMM, (Hochreiter and Mozer, 2001)).

Initially, all state variables have "uncommitted" values then various inputs can trigger the memory state variables to take on values from which the state variables cannot escape – they behave as a memory for the occurrence of an input event. Fig. 7.10 shows the number of updates required to train three models: input output HMM (IOHMM), input output factorial HMM (IOFHMM), and "Memory-based Input-Output Factorial HMM" (MIOFHMM) as a function of the sequence length T.

7.8 Tricks of the Trade

- Sometimes the HMM and its algorithms must be adjusted for bioinformatics applications for example to handle delete states which do not emit symbols in the forward pass.

- HMMs can be used for variable length of the sequences; however care must be take if comparing likelihoods because there are more longer sequences than shorter and the likelihood decreases exponentially with the length
- To deal with small likelihood and probability values is is recommended to compute the values in the log-space
- To avoid zero probabilities for certain sequences which makes certain minima unreachable all probabilities can be kept above a threshold ϵ.
- The EM-algorithm cannot reach probabilities which are exact zero, therefore, as an after-learning postprocessing all small probabilities $\leqslant \epsilon$ can be set to zero. This often helps to generalize from short to very long sequences.
- HMMs are prone to local minima, for example if HMMs are build from unaligned sequences. Global optimization strategies try to avoid theses minima, e.g. deterministic annealing was suggested in the "Userguide" to HMMER version 1.8.

7.9 Profile Hidden Markov Models

Profile Hidden Markov Models code a multiple sequence alignment into an HMM as a position-specific scoring system which can be used to search databases for remote homologous sequences. Figure 7.11 shows a HMM which can be used for homology search. The top row with states indicated with circles are a pattern. The diamond states are inserted strings. The bottom row with states indicated as squares are deletions, where a letter from the pattern is skipped.

To learn an HMM from a set of unaligned positive examples suffers from the problem of local minima. Therefore expensive global optimization strategies must be used to avoid theses minima, e.g. deterministic annealing was suggested in the "Userguide" to HMMER version 1.8. Therefore in most applications an HMM is at least initialized by a multiple alignment of the positive examples.

The use of profile HMMs was made very convenient by the free HMMER package by Sean Eddy (Eddy, 1998) which allows to build and apply HMMs. HMMER supplies a log-odds likelihood of the model compared to a random model to access the significance of the score of a new sequence. Fig. 7.12 shows the architecture of the models used by HMMER. The states indicated by squares and denoted by "Mx" are the consensus string. The circled states denoted by "Dx" are deletion states (non-emitting states), where part of the consensus string can be skipped. The diamond states denoted by "Ix" are insertion states where a substring can be inserted into the consensus string

The other package which enabled a convenient use of HMMs for biological sequences is Sequence Alignment and Modeling system (SAM – `http://www.cse.ucsc.edu/research/compbio/sam.html`) which allows creating, refining, and using HMMs for biological sequence analysis. Also the SAM models represent a refinement of a multiple alignment. Models can be used to both generate multiple alignments and search databases for new members of the family.

Also databases like Protein FAMily database (Pfam) are based on HMMs. 67% of proteins contain at least one Pfam profile HMM and 45% of residues in the protein database are covered in total by the HMMs.

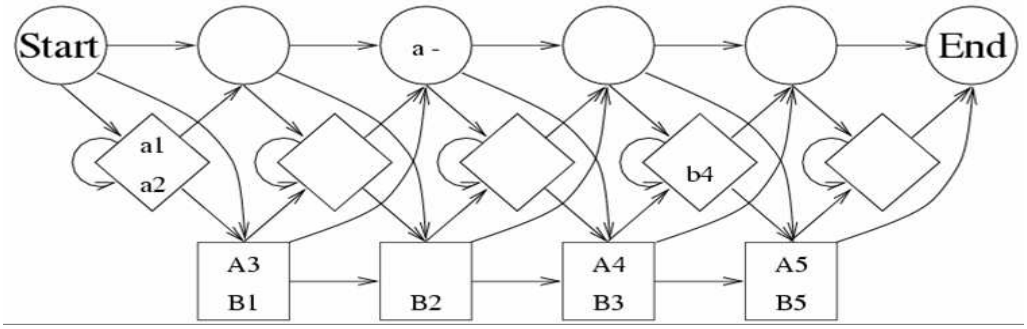


Figure 7.11: Hidden Markov model for homology search. The top row with states indicated with circles are a pattern. The diamond states are inserted strings. The letter from the pattern is skipped.

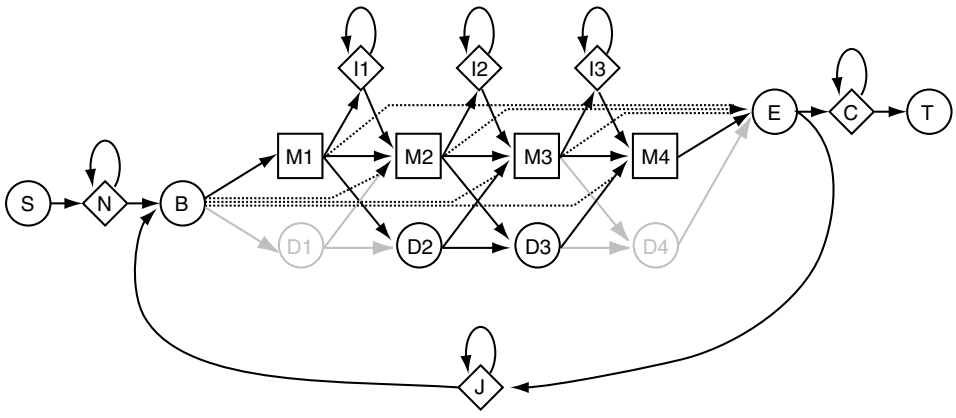


Figure 7.12: The HMMER hidden Markov architecture. The states indicated by squares and denoted by "Mx" form a pattern (consensus string). The circled states denoted by "Dx" are deletion states (non-emitting), where a letter from the pattern can be skipped. The diamond states denoted by "Ix" are insertion states where a substring between letters of the pattern has been inserted. "B" and "E" denote the begin and end state of the pattern, respectively.

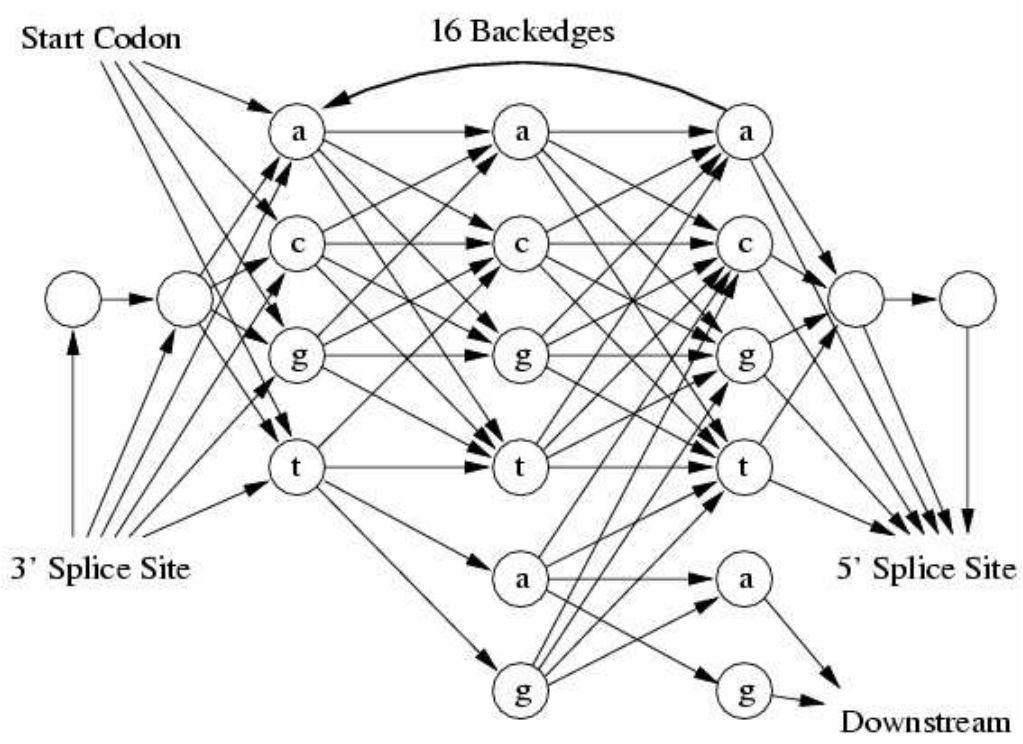


Figure 7.13: An HMM for splice site detection.

Another HMM application which is not associated with profile HMMs is shown in Figure 7.13, where the HMM is used for splice site detection.

Chapter A8

Activation functions

A1 Activation functions for hidden units

A1.1 Sigmoid units: numeric control, soft step function

The commonly used activation functions are sigmoid functions (see A1).

The *logistic sigmoid* function is

$$f(x) \;=\; \frac{1}{1 \;+\; \exp(-\,x)} \tag{A8.1}$$

and the *hyperbolic tangent* activation function ($\tanh$) is

$$f(x) \;=\; \tanh(x) \;=\; \frac{\exp(x) \;-\; \exp(-\,x)}{\exp(x) \;+\; \exp(-\,x)}\,. \tag{A8.2}$$

Both functions are equivalent because

$$\frac{1}{2}\,(\tanh(x/2) \;+\; 1) \;=\; \frac{1}{1 \;+\; \exp(-\,x)}\,. \tag{A8.3}$$

That means through weight scaling and through adjusting the bias weights the networks with logistic function or $\tanh$ can be transformed into each other.

The derivatives of the logistic sigmoid function is the following:

$$f'(x) = f(x)(1 - f(x)), \tag{A8.4}$$

and the derivative of the $\tanh$ activation function is:

$$f'(x) = 1 - f(x)^2. \tag{A8.5}$$

Also quadratic or other activation functions can be used but the fact that the sigmoid functions are squashed into an interval makes learning robust. Because the activation is bounded the derivatives are bounded as we see later. Networks with sigmoid activation are even more robust against unknown input which can drive some activations to regions which are not explored in the training phase.

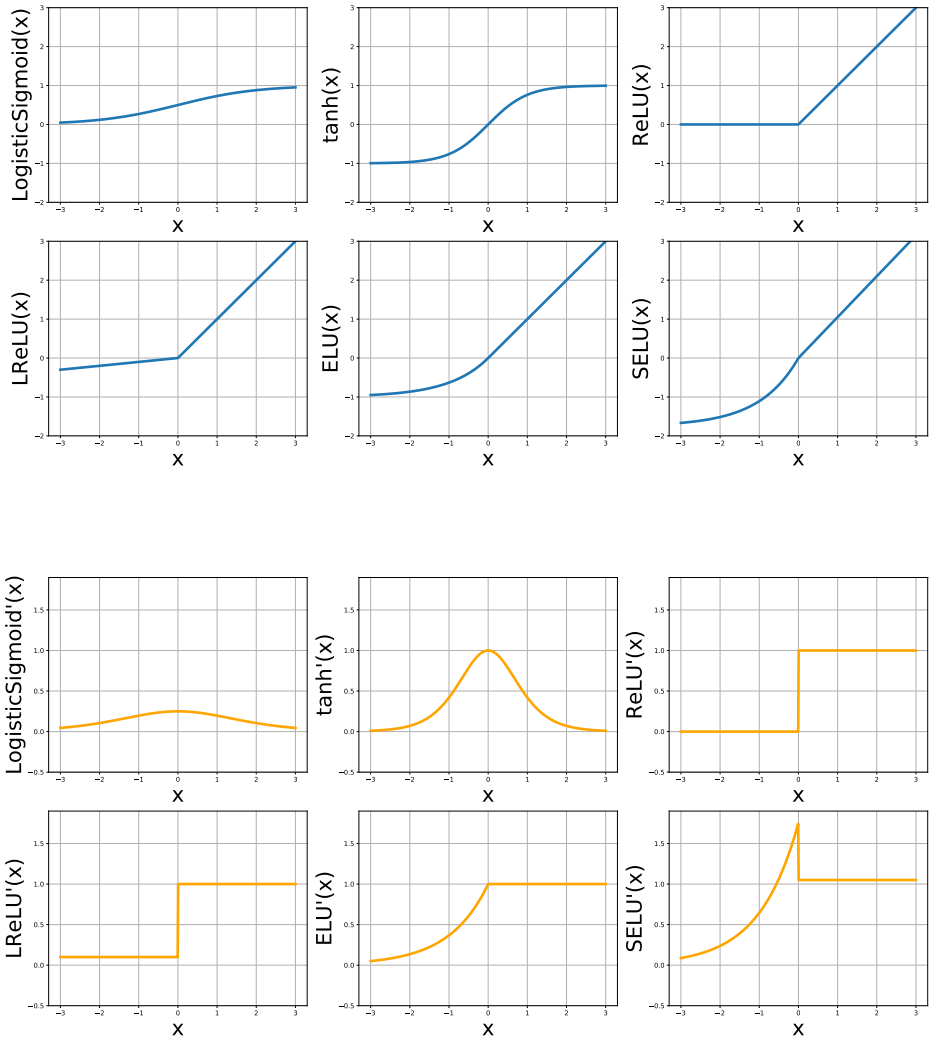


Figure A1: Graphs of commonly used activation functions for hidden layers. **Top:** Graph of the function. **Bottom:** Graph of the derivative.

Symmetric Networks. Symmetric networks can be produced with the $\tanh$ function if the signs of input weights and output weights are changed because $\tanh(-x) = -\tanh(x)$, therefore, $w_2 \tanh(w_1\, x) = (-w_2)\, \tanh((-w_1)\, x)$.

Permutations of the hidden units in one layer leads to equivalent networks. That means the same function can be represented through different network parameters.

Properties:

- Boundedness: Sigmoid units are upper and lower bounded, thus activations stay within the interval $f(x) \in [0, 1]$ for the logistic sigmoid or $f(x) \in [-1, 1]$ for the tanh sigmoid.
- Monotonicity: Both sigmoid functions are strictly monotonically increasing.
- Behavior around origin: The hyperbolic tangent approximates identity near the origin.
- Maximum of absolute derivative: 1 for tanh and 0.25 for logistic sigmoid.

A1.2 Rectified linear units (ReLU): efficient non-linearities

(see A1)

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases} \tag{A8.6}$$

$$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases} \tag{A8.7}$$

Properties:

- Boundedness: ReLU activations are lower bounded, but do not have an upper bound: $f(x) \in [0,)$
- Monotonicity: The ReLU activation function is monotonically increasing.
- Behavior around origin: does not approximate identity around zero.
- Maximum of absolute derivative: $\max_x f'(x) = 1$

A1.3 Leaky rectified linear units (Leaky-ReLU, LReLU)

(see A1)

$$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases} \tag{A8.8}$$

$$f'(x) = \begin{cases} 0.01 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases} \tag{A8.9}$$

Properties:

- Boundedness: LReLU activations are not bounded: $f(x) \in (-\infty, \infty)$
- Monotonicity: The LReLU activation function is strictly monotonically increasing.
- Behavior around origin: does not approximate identity around zero.
- Maximum of absolute derivative: $\max_x f'(x) = 1$

A1.4 (Scaled) exponential linear units (ELU and SELU): countering bias shift and self-normalization

(see A1)

$$f(x) = \lambda \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases} \tag{A8.10}$$

with $\lambda \approx 1.0507$ and $\alpha \approx 1.67326$

$$f'(x) = \lambda \begin{cases} \alpha e^x & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases} \tag{A8.11}$$

Properties:

- Boundedness: SELU activations have a lower bound: $f(x) \in (-\lambda\alpha, \infty)$
- Monotonicity: The SELU activation function is strictly monotonically increasing.
- Behavior around origin: does not approximate identity around zero.
- Maximum of absolute derivative: $\max_x f'(x) = \lambda\alpha$.
- Provide a sufficiently broad network with the ability to keep activations around zero mean and unit variance (self-normalization). See Section on Self-Normalizing Neural Networks.

A1.5 Higher order units.

Higher order units do not use a linear s_i. For example second order units have the form

$$s_i \;=\; \sum_{(j_1,j_2)=(0,0)}^{N} w_{ij_1j_2}\; a_{j_1}\; a_{j_2}\;. \tag{A8.12}$$

Note that linear and constant terms are considered because of the bias unit $a_0 \;=\; 1$.

We will use higher order units in Section 3 in order to gate information flow and to access certain information.

Bibliography

Abrahart, R. J., Anctil, F., Coulibaly, P., Dawson, C. W., Mount, N. J., See, L. M., Shamseldin, A. Y., Solomatine, D. P., Toth, E., and Wilby, R. L. (2012). Two decades of anarchy? Emerging themes and outstanding challenges for neural network river forecasting. *Progress in Physical Geography*, 36(4):480–513.

Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. *Cognitive Science*, 9:147–169.

Almeida, L. B. (1987). A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In *IEEE 1st International Conference on Neural Networks, San Diego*, volume 2, pages 609–618.

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.

Arjona-Medina, J. A., Gillhofer, M., Widrich, M., Unterthiner, T., Brandstetter, J., and Hochreiter, S. (2019). Rudder: Return decomposition for delayed rewards. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 13544–13555. Curran Associates, Inc.

ASCE Task Committee on Application of Artificial Neural Networks (2000). Artificial Neural Networks in Hydrology. Ii: Hydrologic Applications. *Journal Of hydrologic engineering*, pages 124–137.

Back, A. D. and Tsoi, A. C. (1991). FIR and IIR synapses, a new neural network architecture for time series modelling. *Neural Computation*, 3:375–385.

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473. appeared in ICRL 2015.

Baldi, P., Brunak, S., Frasconi, P., Soda, G., and Pollastri, G. (1999). Exploiting the past and the future in protein secondary structure prediction. *Bioinformatics*, 15:937–946.

Barrett, C., Hughey, R., and Karplus, K. (1997). Scoring hidden marov models. *CABIOS*, 13(2):191–19.

Bateman, A., Birney, E., Durbin, R., Eddy, S. R., Howe, K. L., and Sonnhammer, E. L. L. (2000). The Pfam protein families database. *Nucleic Acids Res.*, 28:263–266.

Bateman, A., Coin, L., Durbin, R., Finn, R. D., Hollich, V., Griffiths-Jones, S., Khanna, A., Marshall, M., Moxon, S., Sonnhammer, E. L. L., Studholme, D. J., Yeats, C., and Eddy, S. R. (2004). The pfam protein families database. *Nucleic Acids Research*, 32:D138–141.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

Biswas, A. K. (1970). *History of hydrology*. Elsevier Science Limited.

Bodenhausen, U. (1990). Learning internal representations of pattern sequences in a neural network with adaptive time-delays. In *Proceedings of the International Joint Conference on Neural Networks*. Hillsdale, NJ. Erlbaum.

Bodenhausen, U. and Waibel, A. (1991). The tempo 2 algorithm: Adjusting time-delays by supervised learning. In Lippmann, R. P., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*, pages 155–161. San Mateo, CA: Morgan Kaufmann.

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

Burge, C. and Karlin, S. (1997). Prediction of complete gene structures in human genomic dna. *J. Mol. Biol.*, 268:78–94.

Cheng, J., Dong, L., and Lapata, M. (2016). Long short-term memory-networks for machine reading. *CoRR*, abs/1601.06733.

Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734. Association for Computational Linguistics.

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

Daniell, T. M. (1991). Neural networks. Applications in hydrology and water resources engineering. In *Proceedings of the International Hydrology and Water Resource Symposium*, volume 3, pages 797–802, Perth, Australia. Institution of Engineers.

Daniluk, M., Rocktäschel, T., Welbl, J., and Riedel, S. (2017). Frustratingly short attention spans in neural language modeling. *CoRR*, abs/1702.04521. appeared in ICRL 2017.

de Vries, B. and Principe, J. C. (1991). A theory for neural networks with time delays. In Lippmann, R. P., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*, pages 162–168. San Mateo, CA: Morgan Kaufmann.

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.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Dey, R. and Salemt, F. M. (2017). Gate-variants of gated recurrent unit (gru) neural networks. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1597–1600.

Dumais, S. T. (2004). Latent semantic analysis. *Annual Review of Information Science and Technology*, 38(1):188–230.

Eddy, S. R. (1998). Profile hidden markov models. *Bioinformatics*, 14:755–763.

Eddy, S. R. (2004). What is a hidden markov model? *Nature Biotechnology*, 22:1315–1316.

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

Fan, B., Wang, L., Soong, F. K., and Xie, L. (2015). Photo-real talking head with deep bidirectional lstm. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4884–4888.

Fan, Y., Qian, Y., Xie, F.-L., and Soong, F. K. (2014). Tts synthesis with bidirectional lstm based recurrent neural networks. In *Fifteenth Annual Conference of the International Speech Communication Association*.

Frasconi, P., Gori, M., and Soda, G. (1992). Local feedback multilayered networks. *Neural Computation*, 4:120–130.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202.

Gers, F. A. and Schmidhuber, J. (2000). Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194. IEEE.

Gers, F. A., Schmidhuber, J., and Cummins, F. (2000). Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10).

Ghahramani, Z. and Jordan, M. I. (1996). Factorial hidden markov models. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems 8*, pages 472–478. The MIT Press.

Ghahramani, Z. and Jordan, M. I. (1997). Factorial hidden markov models. *Machine Learning*, 29:245.

Gherrity, M. (1989). A learning algorithm for analog fully recurrent neural networks. In *IEEE/INNS International Joint Conference on Neural Networks, San Diego*, volume 1, pages 643–644.

Gori, M., Bengio, Y., and DeMori, R. (1989). BPS: a learning algorithm for capturing the dynamic nature of speech. In *Proceedings of the International Joint Conference on Neural Networks*, pages 417–423.

Graves, Mohamed, A.-R., and Hinton, G. E. (2013a). Speech recognition with deep recurrent neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649. IEEE.

Graves, A. (2013). Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850.

Graves, A., Fernández, S., and Schmidhuber, J. (2007a). Multi-dimensional recurrent neural networks. In *International Conference on Artificial Neural Networks (ICANN)*, volume 4668 of *Lecture Notes in Computer Science*, pages 6645–6649. Springer, Berlin, Heidelberg.

Graves, A., Fernández, S., and Schmidhuber, J. (2007b). Multi-dimensional recurrent neural networks. *CoRR*, abs/0705.2011.

Graves, A., Liwicki, M., Fernández, S., Bertolami, R., Bunke, H., and Schmidhuber, J. (2008). A novel connectionist system for unconstrained handwriting recognition. *IEEE transactions on pattern analysis and machine intelligence*, 31(5):855–868.

Graves, A., Mohamed, A.-R., and Hinton, G. E. (2013b). Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778.

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2016). Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232.

Gregor, K., Danihelka, I., Graves, A., Rezende, D., and Wierstra, D. (2015a). DRAW: A recurrent neural network for image generation. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1462–1471. PMLR.

Gregor, K., Danihelka, I., Graves, A., and Wierstra, D. (2015b). DRAW: A recurrent neural network for image generation. *CoRR*, abs/1502.04623.

Halff, A. H., Halff, H. M., and Azmoodeh, M. (1993). Predicting runoff from rainfall using neural networks. In *Proceedings of Engineering Hydrology*, pages 760–765, New York, USA.

Hebb, D. O. (1949). *The Organization of Behavior*. Wiley, New York.

Hinton, G. E. and Sejnowski, T. J. (1983). Optimal perceptual inference. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 448–453. Citeseer.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Tech. Univ. München.

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

Hochreiter, S., Heusel, M., and Obermayer, K. (2007a). Fast model-based protein homology detection without alignment. *Bioinformatics*, 23(14):1728–1736.

Hochreiter, S., Heusel, M., and Obermayer, K. (2007b). Fast model-based protein homology detection without alignment. *Bioinformatics*, 23(14):1728–1736.

Hochreiter, S. and Mozer, M. C. (2001). A discrete probabilistic memory model for discovering dependencies in time. In Dorffner, G., Bischof, H., and Hornik, K., editors, *International Conference on Artificial Neural Networks*, pages 661–668. Springer.

Hochreiter, S. and Schmidhuber, J. (1995). Long short-term memory. Technical Report FKI-207-95, Fakultät für Informatik, Technische Universität München. Revised 1996 (see www.idsia.ch/˜juergen, www7.informatik.tu-muenchen.de/˜hochreit).

Hochreiter, S. and Schmidhuber, J. (1996). Bridging long time lags by weight guessing and "Long Short-Term Memory". In Silva, F. L., Principe, J. C., and Almeida, L. B., editors, *Spatiotemporal models in biological and artificial systems*, pages 65–72. IOS Press, Amsterdam, Netherlands. Serie: Frontiers in Artificial Intelligence and Applications, volume 37.

Hochreiter, S. and Schmidhuber, J. (1997a). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Hochreiter, S. and Schmidhuber, J. (1997b). LSTM can solve hard long time lag problems. In Mozer, M. C., Jordan, M. I., and Petsche, T., editors, *Advances in Neural Information Processing Systems 9*, pages 473–479. MIT Press, Cambridge MA.

Hochreiter, S. and Schmidhuber, J. (1997c). Unsupervised coding with Lococode. In Gerstner, W., Germond, A., Hasler, M., and Nicoud, J.-D., editors, *Proceedings of the International Conference on Artificial Neural Networks, Lausanne, Switzerland*, pages 655–660. Springer.

Hochreiter, S., Younger, A. S., and Conwell, P. R. (2001). Learning to learn using gradient descent. In Dorffner, G., Bischof, H., and Hornik, K., editors, *International Conference on Artificial Neural Networks*, pages 87–94. Springer.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proc. of the National Academy of Sciences*, 79:2554–2558.

Hossain, M., Sohel, F., Shiratuddin, M. F., and Laga, H. (2019). A comprehensive survey of deep learning for image captioning. *ACM Computing Surveys (CSUR)*, 51(6):118.

Hutter, M. (2012). The human knowledge compression contest. *URL http://prize. hutter1. net*, 6.

Jaeger, H. (2002). *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach*, volume 5. GMD-Forschungszentrum Informationstechnik Bonn.

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.

Kalchbrenner, N., Danihelka, I., and Graves, A. (2015). Grid long short-term memory. *arXiv*, 1507.01526.

Karpathy, A. and Fei-Fei, L. (2015). Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3128–3137.

Karplus, K., Barrett, C., Cline, M., Diekhans, M., Grate, L., and Hughey, R. (1999). Predicting protein structure using only sequence information. *Proteins: Structure, Function, and Genetics*, 37(S3):121–125.

Karplus, K., Barrett, C., and Hughey, R. (1998). Hidden Markov models for detecting remote protein homologies. *Bioinformatics*, 14(10):846–856.

Kiros, R., Salakhutdinov, R., and Zemel, R. S. (2014). Unifying visual-semantic embeddings with multimodal neural language models. *arXiv preprint arXiv:1411.2539*.

Koutsoyiannis, D., Kundzewicz, Z. W., Watkins, F., and Gardner, C. (2010). Something old, something new, something red, something blue.

Kratzert, F., Klotz, D., Brenner, C., Schulz, K., and Herrnegger, M. (2018). Rainfall–runoff modelling using long short-term memory (lstm) networks. *Hydrology and Earth System Sciences*, 22(11):6005–6022.

Kratzert, F., Klotz, D., Sampson, A. K., Hochreiter, S., Nearing, G., et al. (2019a). Prediction in ungauged basins with long short-term memory networks. *EarthArXiv preprint 10.31223/osf.io/4rysp*.

Kratzert, F., Klotz, D., Shalev, G., Klambauer, G., Hochreiter, S., and Nearing, G. (2019b). Benchmarking a catchment-aware long short-term memory network (lstm) for large-scale hydrological modeling. *arXiv preprint arXiv:1907.08456*.

Krause, B., Lu, L., Murray, I., and Renals, S. (2016). Multiplicative lstm for sequence modelling. *arXiv preprint arXiv:1609.07959*.

Krogh, A. (1997). Two methods for improving performance of a hmm and their application for gene finding. In Gaasterland, T., Karp, P., Karplus, K., Ouzounis, C., Sander, C., and Valencia, A., editors, *Proceedings of the Fifth International Conference on Intelligent Systems for Molecular Biology*, pages 179–186. AAAI Press, Menlo Park, CA.

Krogh, A., Brown, M., Mian, I., Sjölander, K., and Haussler, D. (1994a). Hidden Markov models in computational biology: Applications to protein modeling. *Journal of Molecular Biology*, 235:1501–1531.

Krogh, A., Mian, I. S., and Haussler, D. (1994b). A hidden Markov model that finds genes in E. coli DNA. *Nucl. Acids Res.*, 22:4768–4778.

Krueger, D., Maharaj, T., Kramár, J., Pezeshki, M., Ballas, N., Ke, N. R., Goyal, A., Bengio, Y., Courville, A. C., and Pal, C. J. (2017). Zoneout: Regularizing rnns by randomly preserving hidden activations. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

Kulp, D., Haussler, D., Reese, M. G., and Eeckman, F. H. (1996). A generalized hidden markov model for the recognition of human genes in dna. In States, D. J., Agarwal, P., Gaasterland, T., Hunter, L., and Smith, R. F., editors, *Proceedings of the Fourth International Conference on Intelligent Systems for Molecular Biology*, pages 134–142. AAAI Press, Menlo Park, CA.

Le, Q. V., Jaitly, N., and Hinton, G. E. (2015). A simple way to initialize recurrent networks of rectified linear units. *CoRR*, abs/1504.00941.

LeCun, Y. (1989). Generalization and network design strategies. In *Connectionism in perspective*, volume 19. Citeseer.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278–2324.

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.

Lin, Z., Feng, M., dos Santos, C. N., Yu, M., Xiang, B., Zhou, B., and Bengio, Y. (2017). A structured self-attentive sentence embedding. *CoRR*, abs/1703.03130. appeared in ICRL 2017.

Lio, P., Thorne, J. L., Goldman, N., and Jones, D. T. (1999). Passml: Combining evolutionary inference and protein secondary structure prediction. *Bioinformatics*, 14:726–73.

Liwicki, M. and Bunke, H. (2005). Iam-ondb-an on-line english sentence database acquired from handwritten text on a whiteboard. In *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, pages 956–961. IEEE.

Lotter, W., Kreiman, G., and Cox, D. (2016). Deep predictive coding networks for video prediction and unsupervised learning. *arXiv preprint arXiv:1605.08104*.

Luong, M.-T., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025.

Marçais, J. and de Dreuzy, J. R. (2017). Prospective Interest of Deep Learning for Hydrological Inference. *Groundwater*, 55(5):688–692.

Marcus, M. P. and Marcinkiewicz, M. A. (1993). Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330.

Matthews, M. B. (1990). Neural network nonlinear adaptive filtering using the extended Kalman filter algorithm. In *Proceedings of the Interantional Neural Networks Conference*, pages 115–119.

McAuley, J., Pandey, R., and Leskovec, J. (2015). Inferring networks of substitutable and complementary products. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 785–794. ACM.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T., Yih, W.-t., and Zweig, G. (2013b). Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, Atlanta, Georgia. Association for Computational Linguistics.

Mozer, M. C. (1989). A focused back-propagation algorithm for temporal sequence recognition. *Complex Systems*, 3:349–381.

Mulvaney, T. J. (1850). On the use of self-registering rain and flood gauges in making observations of the relations of rainfall and of flood discharges in a given catchment. In *Proceedings Institution of Civil Engineers*, volume 4, pages 18–31.

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

Narendra, K. S. and Parthasarathy, K. (1990). Identification and control of dynamical systems using neural networks. In *IEEE Transactions on Neural Networks*, volume 1, pages 4–27.

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

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics.

Park, J., Karplus, K., Barrett, C., Hughey, R., Haussler, D., Hubbard, T., and Chothia, C. (1998). Sequence comparisons using multiple sequences detect three times as many remote homologues as pairwise methods. *Journal of Molecular Biology*, 284(4):1201–1210.

Pearlmutter, B. A. (1989). Learning state space trajectories in recurrent neural networks. *Neural Computation*, 1(2):263–269.

Pearlmutter, B. A. (1995). Gradient calculations for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural Networks*, 6(5):1212–1228.

Pennington, J., Socher, R., and Manning, C. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.

Pineda, F. J. (1987). Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 19(59):2229–2232.

Puskorius, G. V. and Feldkamp, L. A. (1994). Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks. *IEEE Transactions on Neural Networks*, 5(2):279–297.

Quang, D. and Xie, X. (2016). Danq: a hybrid convolutional and recurrent deep neural network for quantifying the function of dna sequences. *Nucleic acids research*, 44(11):e107–e107.

Radford, A., Jozefowicz, R., and Sutskever, I. (2017). Learning to generate reviews and discovering sentiment. *arXiv preprint arXiv:1704.01444*.

Rao, C. R. and Toutenburg, H. (1999). *Linear Models — Least Squares and Alternatives*. Springer Series in Statistics. Springer, New York, Berlin, Heidelberg, London, Tokyo, 2 edition. ISBN 0-387-98848-3.

Remesan, R. and Mathew, J. (2014). *Hydrological data driven modelling: a case study approach*, volume 1. Springer.

Robinson, A. J. and Fallside, F. (1987). The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Cambridge University Engineering Department.

Salton, G. (1962). Some experiments in the generation of word and document associations. In *Proceedings of the December 4-6, 1962, Fall Joint Computer Conference*, AFIPS '62 (Fall), pages 234–250, New York, NY, USA. ACM.

Salton, G., Wong, A., and Yang, C. S. (1975). A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620.

Schmidhuber, J. (1991). An $O(n^3)$ learning algorithm for fully recurrent networks. Technical Report FKI-151-91, Institut für Informatik, Technische Universität München.

Schmidhuber, J. (1992). A fixed size storage $O(n^3)$ time complexity learning algorithm for fully recurrent continually running networks. *Neural Computation*, 4(2):243–248.

Schultz, J., Copley, R. R., Doerks, T., Ponting, C. P., and Bork, P. (2000). A web-based tool for the study of genetically mobile domains. *Nucleic Acids Res.*, 28:231–23.

Schuster, M., Paliwal, K. K., and General, A. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*.

Shen, C. (2017). A trans-disciplinary review of deep learning research for water resources scientists. *arXiv preprint arXiv:1712.02162*.

Shi, X., Chen, Z., Wang, H., Yeung, D., Wong, W., and Woo, W. (2015). Convolutional lstm network: A machine learning approach for precipitation nowcasting. In *Advances in neural information processing systems*, pages 802–810.

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.

Sjölander, K., Karplus, K., Brown, M., Hughey, R., Krogh, A., Mian, I. S., and Haussler, D. (1996). Dirichlet mixtures: a method for improved detection of weak but significant protein sequence homology. *CABIOS*, 12(4):327–345.

Solomatine, D., See, L. M., and Abrahart, R. J. (2009). Data-driven modelling: concepts, approaches and experiences. In *Practical hydroinformatics*, pages 17–30. Springer.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958.

Stacks Project Authors, T. (2018). *Stacks Project*. `https://stacks.math.columbia.edu`.

Stollenga, M., Byeon, W., Liwicki, M., and Schmidhuber, J. (2015). Parallel multi-dimensional lstm, with application to fast biomedical volumetric image segmentation. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS)*, pages 2998–3006.

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.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc.

Suwajanakorn, S., Seitz, S. M., and Kemelmacher-Shlizerman, I. (2017). Synthesizing obama: learning lip sync from audio. *ACM Transactions on Graphics (TOG)*, 36(4):95.

Tallec, C. and Ollivier, Y. (2018a). Can recurrent neural networks warp time? In *International Conference on Learning Representations*.

Tallec, C. and Ollivier, Y. (2018b). Unbiased online recurrent optimization. In *International Conference on Learning Representations*.

Vapnik, V. N. (1998). *Statistical Learning Theory*. Adaptive and learning systems for signal processing, communications, and control. Wiley, New York.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017a). Attention is all you need. *CoRR*, abs/1706.03762.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017b). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc.

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*.

Wan, E. A. (1990). Temporal backpropagation for FIR neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, volume 1, pages 575–580.

Wang, P., Qian, Y., Soong, F. K., He, L., and Zhao, H. (2015). Part-of-speech tagging with bidirectional long short-term memory recurrent neural network. *arXiv preprint arXiv:1510.06168*.

Weiss, G., Goldberg, Y., and Yahav, E. (2018). On the practical computational power of finite precision RNNs for language recognition. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 740–745, Melbourne, Australia. Association for Computational Linguistics.

Werbos, P. J. (1981). Applications of advances in nonlinear sensitivity analysis. In Drenick, R. F. and Kozin, F., editors, *System Modeling and Optimization: Proceedings of the 10th IFIP Conference, 31.8 - 4.9, NYC*, pages 762–770. Springer-Verlag. number 38 in Lecture Notes in Control and Information Sciences.

Werbos, P. J. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1.

Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

Whittle, P. (1951). *Hypothesis Testing in Time Series Analysis*. Uppsala, Almqvist & Wiksells boktr.

Williams, R. J. (1992). Training recurrent networks using the extended Kalman filter. In *Proceedings of the International Joint Conference on Neural Networks IJCNN-92*, pages 241–250. Lawrence Erlbaum, Hillsdale, N.J.

Williams, R. J. and Peng, J. (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 4:491–501.

Williams, R. J. and Zipser, D. (1989). A learning algorithm for continually running fully recurrent networks. *Neural Computation*, 1(2):270–280.

Williams, R. J. and Zipser, D. (1992). Gradient-based learning algorithms for recurrent networks and their computational complexity. In Chauvin, Y. and Rumelhart, D. E., editors, *Backpropagation: Theory, Architectures and Applications*. Hillsdale, NJ: Erlbaum.

Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A. C., Salakhutdinov, R., Zemel, R. S., and Bengio, Y. (2015a). Show, attend and tell: Neural image caption generation with visual attention. *CoRR*, abs/1502.03044.

Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A. C., Salakhutdinov, R., Zemel, R. S., and Bengio, Y. (2015b). Show, attend and tell: Neural image caption generation with visual attention. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2048–2057.

Young, P. C. and Beven, K. J. (1994). Data-based mechanistic modelling and the rainfall-flow non-linearity. *Environmetrics*, 5(3):335–363.

Zaremba, W., Sutskever, I., and Vinyals, O. (2014). Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.

Zhu, M. and Fujita, M. (1993). *Application of neural networks to runoff forecast*, volume 3. Springer, Dodrecht.