February 26-March 1: Advanced machine learning and data analysis for the physical sciences

Morten Hjorth-Jensen[1,2]

Department of Physics and Center for Computing in Science Education, University of Oslo, Norway[1]

Department of Physics and Astronomy and Facility for Rare Isotope Beams, Michigan State University, East Lansing, Michigan, USA[2]

February 26-March 1, 2024

Plans for the week February 26-March 1

1. Finalizing discussion of Convolutional Neural Networks (CNNs)
2. Discussion of recurrent neural networks (RNNs)
3. Video of lecture
4. Whiteboard notes
5. Reading recommendations:
   - 5.1 Goodfellow, Bengio and Courville's chapter 10 from Deep Learning
   - 5.2 Sebastian Rashcka et al, chapter 15, Machine learning with Sickit-Learn and PyTorch
   - 5.3 David Foster, Generative Deep Learning with TensorFlow, see chapter 5

The last two books have codes for RNNs in PyTorch and TensorFlow/Keras. Next week we will study the solution of differential equations.

From FFNNs and CNNs to recurrent neural networks (RNNs)

There are limitation of FFNNs, one of which being that FFNNs are not designed to handle sequential data (data for which the order matters) effectively because they lack the capabilities of storing information about previous inputs; each input is being treated indepen- dently. This is a limitation when dealing with sequential data where past information can be vital to correctly process current and future inputs.

Feedback connections

In contrast to FFNNs, recurrent networks introduce feedback connections, meaning the information is allowed to be carried to subsequent nodes across different time steps. These cyclic or feedback connections have the objective of providing the network with some kind of memory, making RNNs particularly suited for time- series data, natural language processing, speech recognition, and several other problems for which the order of the data is crucial. The RNN architectures vary greatly in how they manage information flow and memory in the network.

Vanishing gradients

Different architectures often aim at improving some sub-optimal characteristics of the network. The simplest form of recurrent network, commonly called simple or vanilla RNN, for example, is known to suffer from the problem of vanishing gradients. This problem arises due to the nature of backpropagation in time. Gradients of the cost/loss function may get exponentially small (or large) if there are many layers in the network, which is the case of RNN when the sequence gets long.

Recurrent neural networks (RNNs): Overarching view

Till now our focus has been, including convolutional neural networks as well, on feedforward neural networks. The output or the activations flow only in one direction, from the input layer to the output layer.

A recurrent neural network (RNN) looks very much like a feedforward neural network, except that it also has connections pointing backward.

RNNs are used to analyze time series data such as stock prices, and tell you when to buy or sell. In autonomous driving systems, they can anticipate car trajectories and help avoid accidents. More generally, they can work on sequences of arbitrary lengths, rather than on fixed-sized inputs like all the nets we have discussed so far. For example, they can take sentences, documents, or audio samples as input, making them extremely useful for natural language processing systems such as automatic translation and speech-to-text.

Sequential data only?

An important issue is that in many deep learning methods we assume that the input and output data can be treated as independent and identically distributed, normally abbreviated to **iid**. This means that the data we use can be seen as mutually independent.
This is however not the case for most data sets used in RNNs since we are dealing with sequences of data with strong inter-dependencies. This applies in particular to time series, which are sequential by contruction.

Differential equations

As an example, the solutions of ordinary differential equations can be represented as a time series, similarly, how stock prices evolve as function of time is another example of a typical time series, or voice records and many other examples.
Not all sequential data may however have a time stamp, texts being a typical example thereof, or DNA sequences.
The main focus here is on data that can be structured either as time series or as ordered series of data. We will not focus on for example natural language processing or similar data sets.

A simple example

```
# Start importing packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Dense, SimpleRNN, LSTM, GRU
from tensorflow.keras import optimizers
from tensorflow.keras import regularizers
from tensorflow.keras.utils import to_categorical



# convert into dataset matrix
def convertToMatrix(data, step):
 X, Y =[], []
 for i in range(len(data)-step):
  d=i+step
  X.append(data[i:d,])
  Y.append(data[d,])
 return np.array(X), np.array(Y)

step = 4
N = 1000
Tp = 800
```

RNNs

RNNs are very powerful, because they combine two properties:

1. Distributed hidden state that allows them to store a lot of information about the past efficiently.
2. Non-linear dynamics that allows them to update their hidden state in complicated ways.

With enough neurons and time, RNNs can compute anything that can be computed by your computer.

What kinds of behaviour can RNNs exhibit?

1. They can oscillate.
2. They can settle to point attractors.
3. They can behave chaotically.
4. RNNs could potentially learn to implement lots of small programs that each capture a nugget of knowledge and run in parallel, interacting to produce very complicated effects.

But the extensive computational needs of RNNs makes them very hard to train.

Basic layout, Figures from Sebastian Rashcka et al, Machine learning with Sickit-Learn and PyTorch

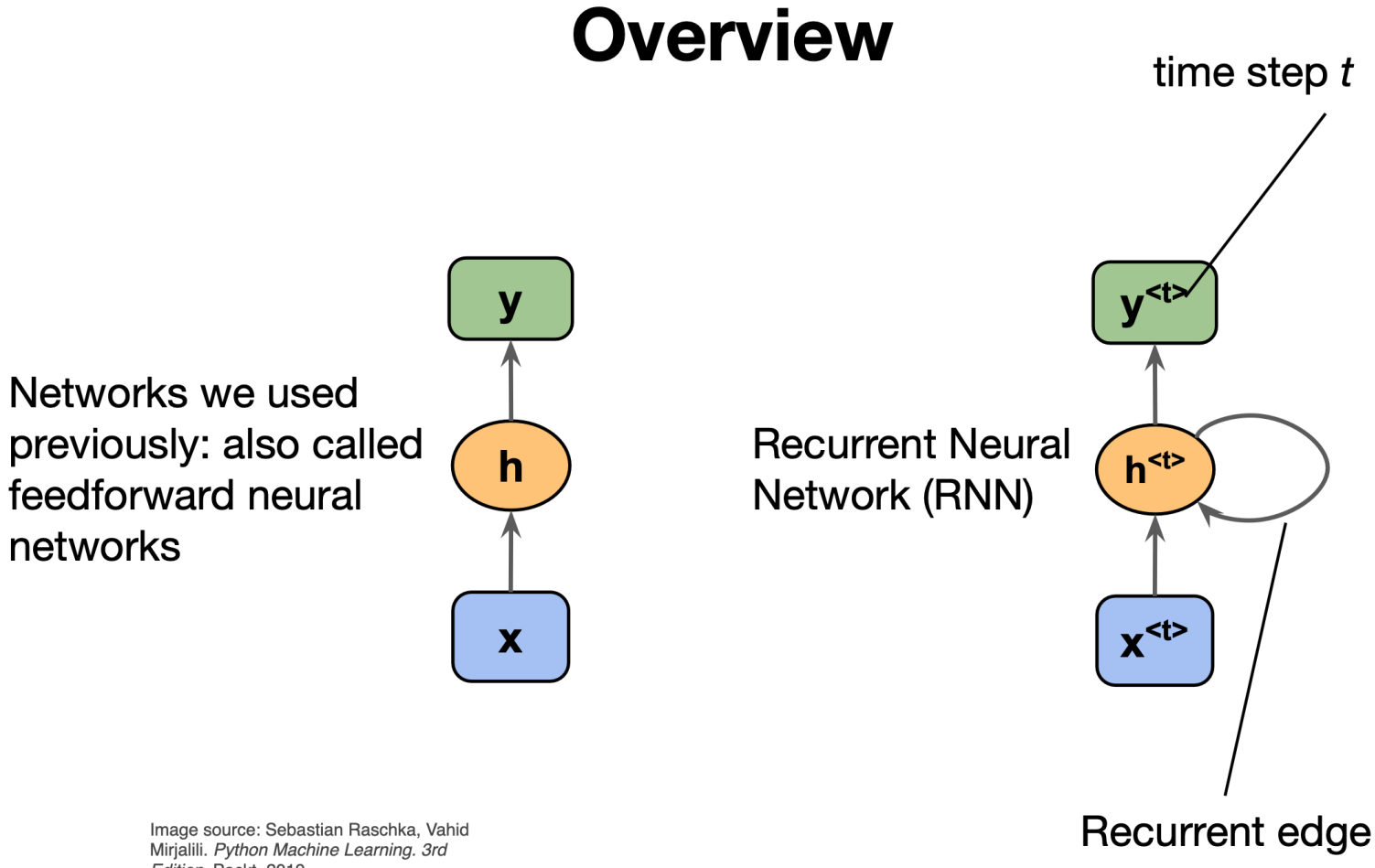


Image source: Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning. 3rd Edition.* Packt, 2019

Solving differential equations with RNNs

To gain some intuition on how we can use RNNs for time series, let us tailor the representation of the solution of a differential equation as a time series.
Consider the famous differential equation (Newton's equation of motion for damped harmonic oscillations, scaled in terms of dimensionless time)

$$\frac{d^2x}{dt^2} + \eta\frac{dx}{dt} + x(t) = F(t),$$

where η is a constant used in scaling time into a dimensionless variable and $F(t)$ is an external force acting on the system. The constant η is a so-called damping.

Two first-order differential equations

In solving the above second-order equation, it is common to rewrite it in terms of two coupled first-order equations with the velocity

$$v(t) = \frac{dx}{dt},$$

and the acceleration

$$\frac{dv}{dt} = F(t) - \eta v(t) - x(t).$$

With the initial conditions $v_0 = v(t_0)$ and $x_0 = x(t_0)$ defined, we can integrate these equations and find their respective solutions.

Velocity only

Let us focus on the velocity only. Discretizing and using the simplest possible approximation for the derivative, we have Euler's forward method for the updated velocity at a time step $i+1$ given by

$$v_{i+1} = v_i + \Delta t \frac{dv}{dt}_{|_{v=v_i}} = v_i + \Delta t \left(F_i - \eta v_i - x_i \right).$$

Defining a function

$$h_i(x_i, v_i, F_i) = v_i + \Delta t \left(F_i - \eta v_i - x_i \right),$$

we have

$$v_{i+1} = h_i(x_i, v_i, F_i).$$

Linking with RNNs

The equation

$$v_{i+1} = h_i(x_i, v_i, F_i).$$

can be used to train a feed-forward neural network with inputs v_i and outputs v_{i+1} at a time t_i. But we can think of this also as a recurrent neural network with inputs v_i, x_i and F_i at each time step t_i, and producing an output v_{i+1}.
Noting that

$$v_i = v_{i-1} + \Delta t \left(F_{i-1} - \eta v_{i-1} - x_{i-1}\right) = h_{i-1}.$$

we have

$$v_i = h_{i-1}(x_{i-1}, v_{i-1}, F_{i-1}),$$

and we can rewrite

$$v_{i+1} = h_i(x_i, h_{i-1}, F_i).$$

Minor rewrite

We can thus set up a recurring series which depends on the inputs x_i and F_i and the previous values h_{i-1}. We assume now that the inputs at each step (or time t_i) is given by x_i only and we denote the outputs for $\tilde{y}_i$ instead of v_{i_1}, we have then the compact equation for our outputs at each step t_i

$$y_i = h_i(x_i, h_{i-1}).$$

We can think of this as an element in a recurrent network where our network (our model) produces an output y_i which is then compared with a target value through a given cost/loss function that we optimize. The target values at a given step t_i could be the results of a measurement or simply the analytical results of a differential equation.

RNNs in more detail

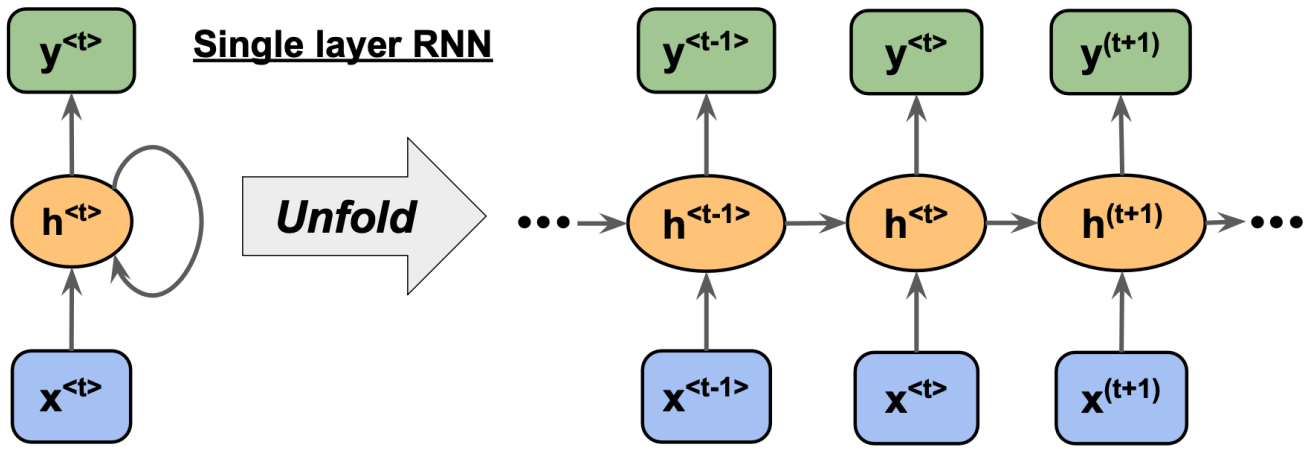
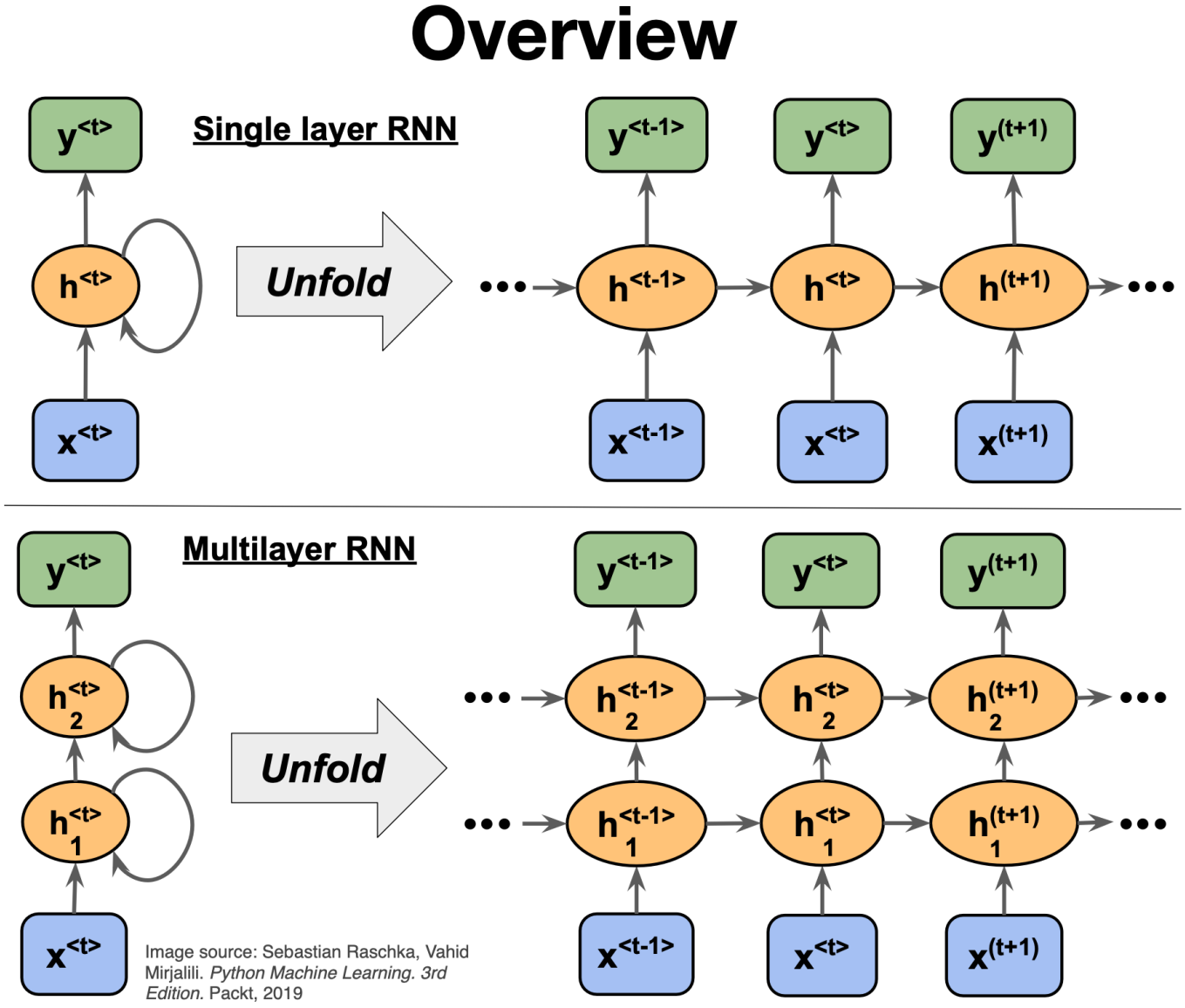
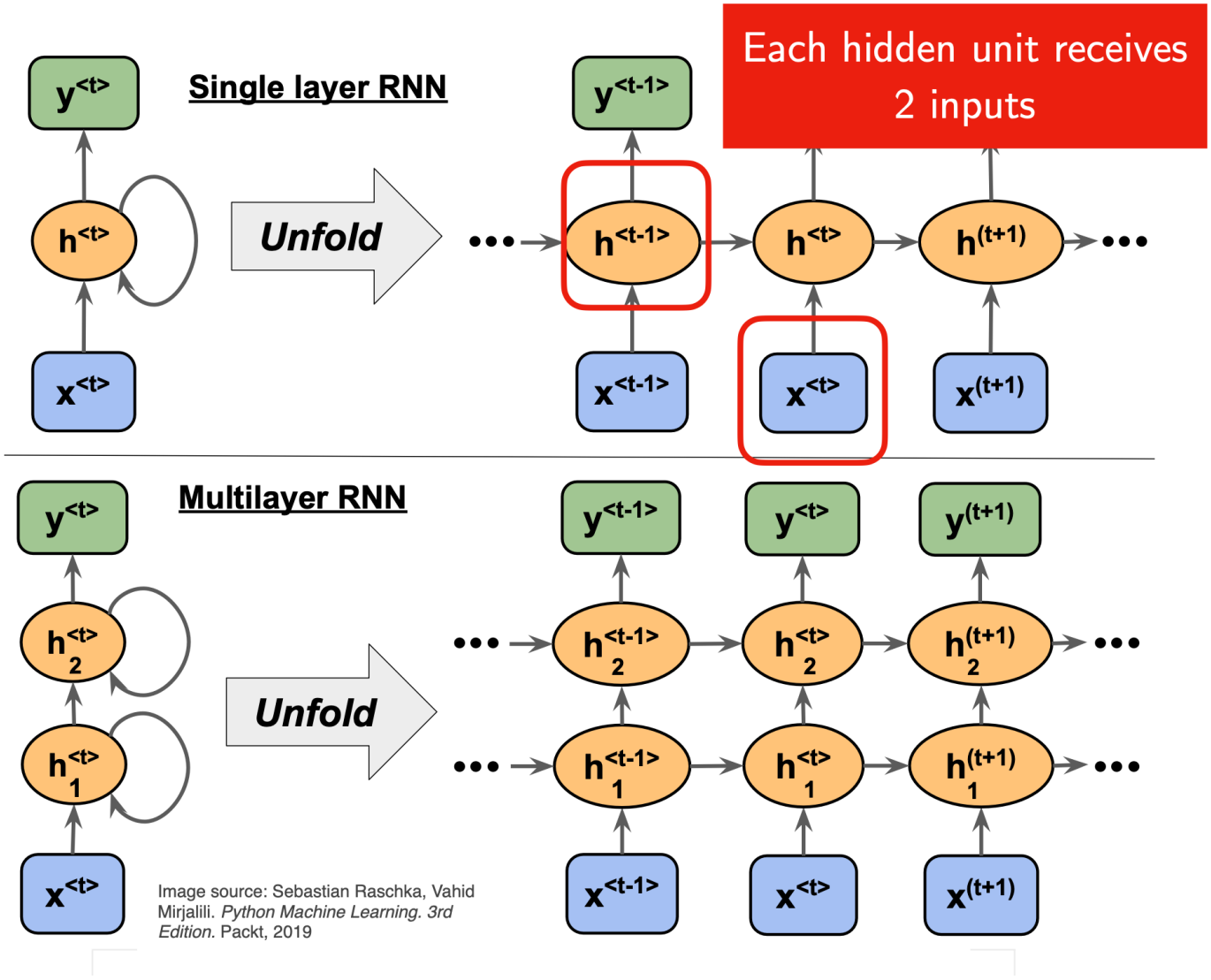


Image source: Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning. 3rd Edition.* Packt, 2019

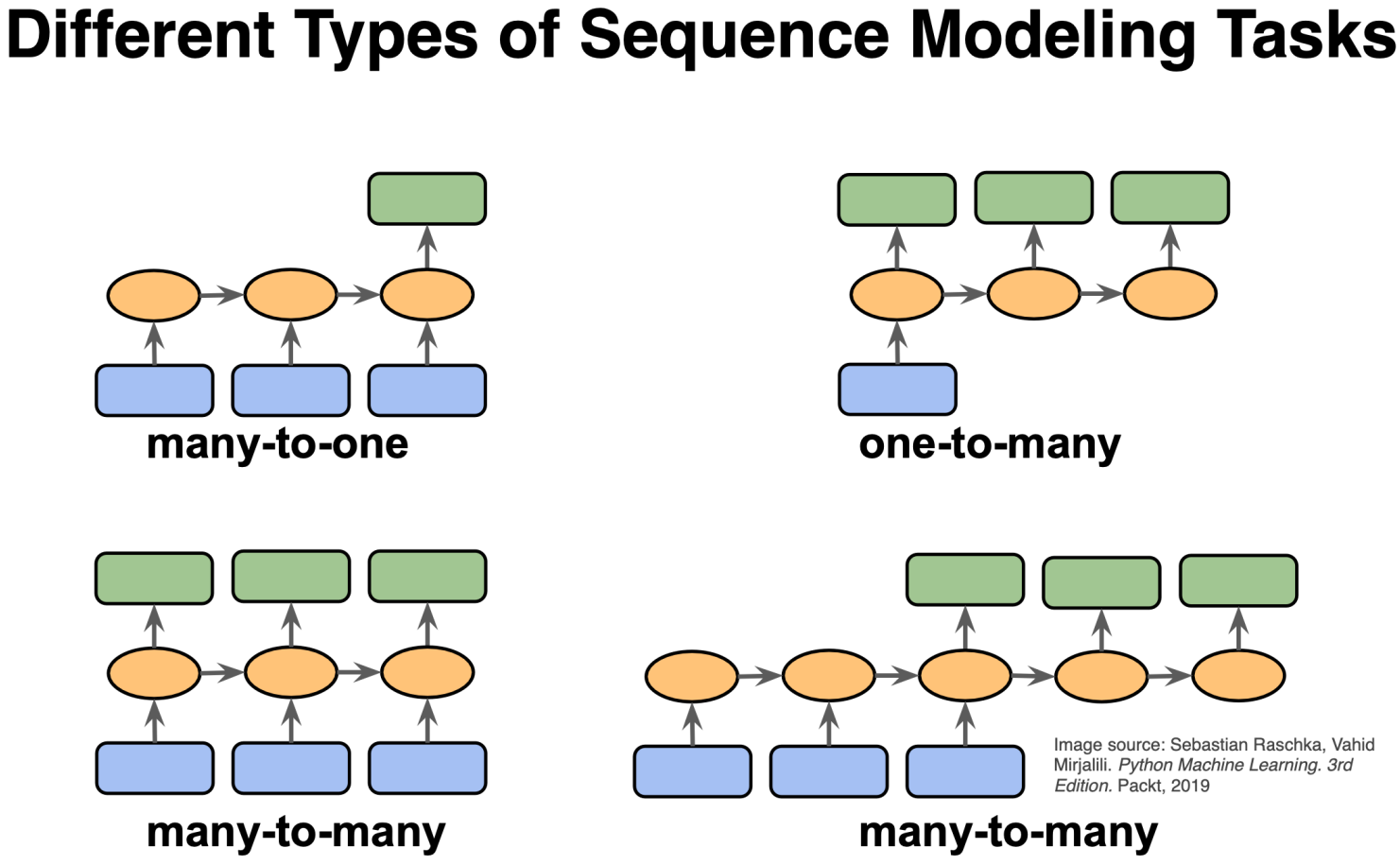
RNNs in more detail, part 2



RNNs in more detail, part 3



RNNs in more detail, part 4



RNNs in more detail, part 5

Weight matrices in a single-hidden layer RNN

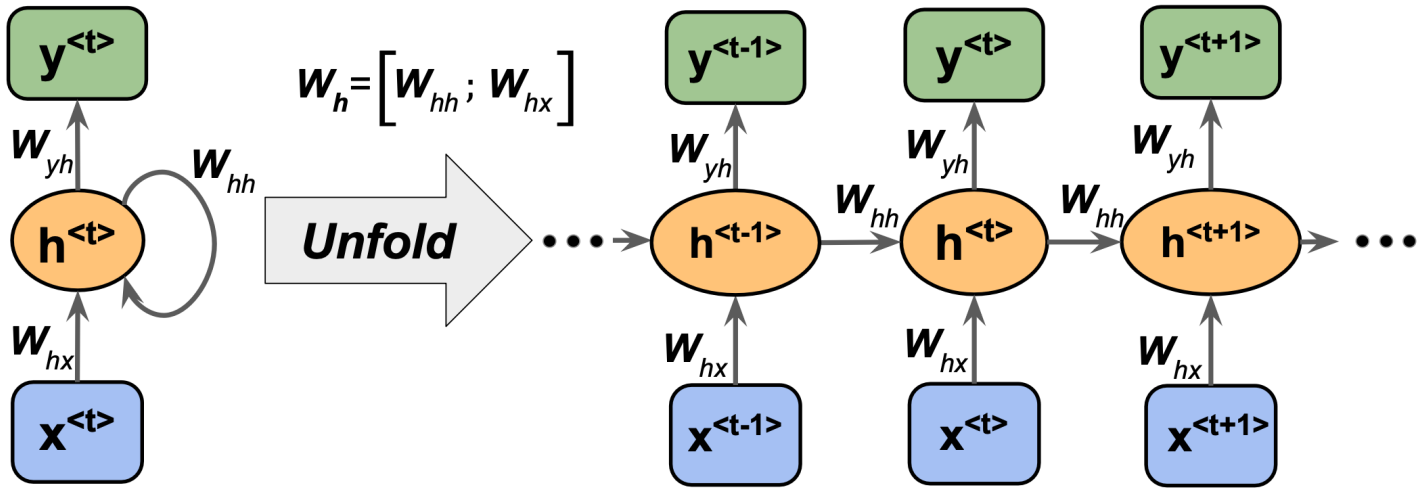


Image source: Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning. 3rd Edition.* Packt, 2019

RNNs in more detail, part 6

Weight matrices in a single-hidden layer RNN

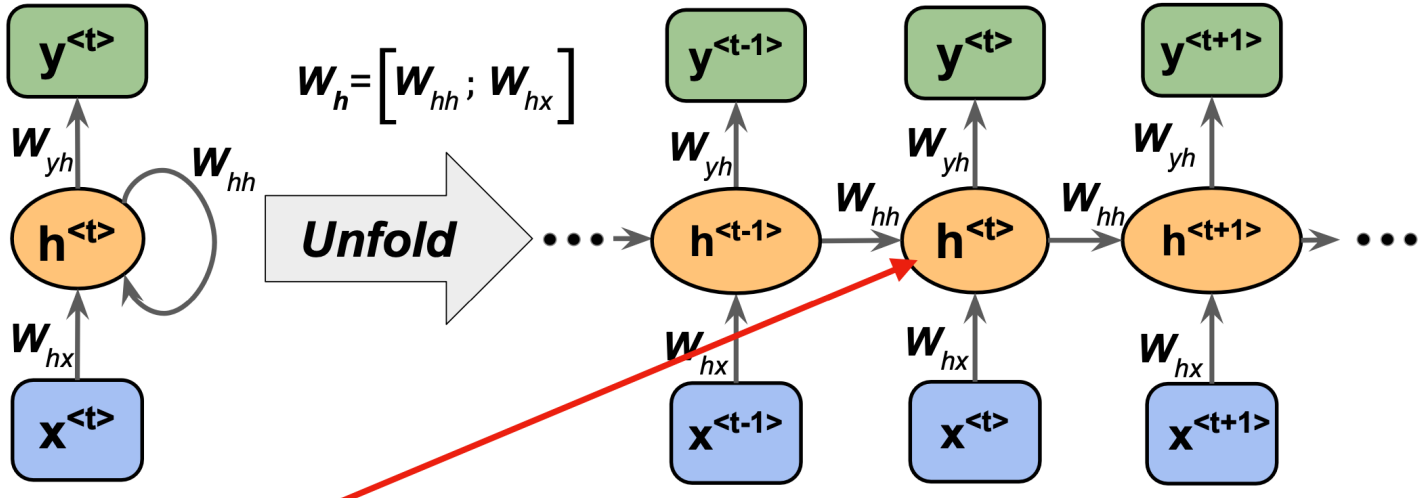


Image source: Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning. 3rd Edition.* Packt, 2019

Net input:

$$\mathbf{z}_h^{\langle t \rangle} = \mathbf{W}_{hx}\mathbf{x}^{\langle t \rangle} + \mathbf{W}_{hh}\mathbf{h}^{\langle t-1 \rangle} + \mathbf{b}_h$$

Activation:

$$\mathbf{h}^{\langle t \rangle} = \sigma_h\left(\mathbf{z}_h^{\langle t \rangle}\right)$$

RNNs in more detail, part 7

Weight matrices in a single-hidden layer RNN

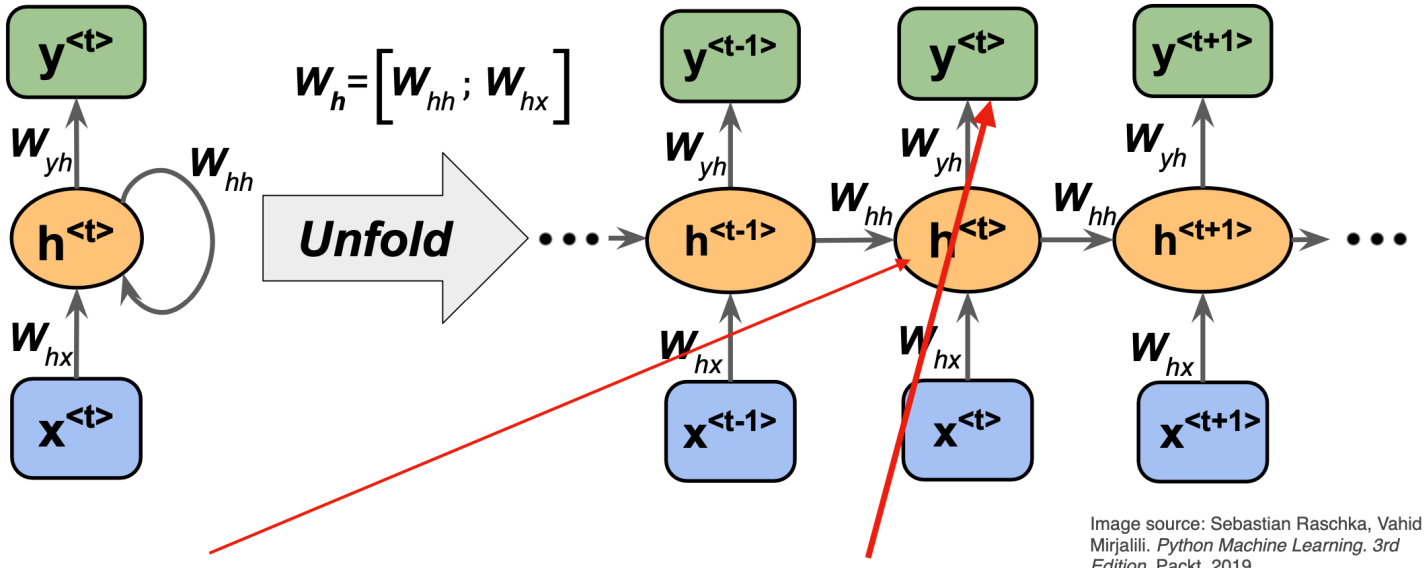


Image source: Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning. 3rd Edition.* Packt, 2019

Net input:

$$\mathbf{z}_h^{\langle t \rangle} = \mathbf{W}_{hx}\mathbf{x}^{\langle t \rangle} + \mathbf{W}_{hh}\mathbf{h}^{\langle t-1 \rangle} + \mathbf{b}_h$$

Activation:

$$\mathbf{h}^{\langle t \rangle} = \sigma_h\left(\mathbf{z}_h^{\langle t \rangle}\right)$$

Net input:

$$\mathbf{z}_y^{\langle t \rangle} = \mathbf{W}_{yh}\mathbf{h}^{\langle t \rangle} + \mathbf{b}_y$$

Output:

$$\mathbf{y}^{\langle t \rangle} = \sigma_y\left(\mathbf{z}_y^{\langle t \rangle}\right)$$

Backpropagation through time

We can think of the recurrent net as a layered, feed-forward net with shared weights and then train the feed-forward net with weight constraints.

We can also think of this training algorithm in the time domain:

1. The forward pass builds up a stack of the activities of all the units at each time step.
2. The backward pass peels activities off the stack to compute the error derivatives at each time step.
3. After the backward pass we add together the derivatives at all the different times for each weight.

The backward pass is linear

1. There is a big difference between the forward and backward passes.
2. In the forward pass we use squashing functions (like the logistic) to prevent the activity vectors from exploding.
3. The backward pass, is completely linear. If you double the error derivatives at the final layer, all the error derivatives will double.

The forward pass determines the slope of the linear function used for backpropagating through each neuron

The problem of exploding or vanishing gradients

- What happens to the magnitude of the gradients as we backpropagate through many layers?
  1. If the weights are small, the gradients shrink exponentially.
  2. If the weights are big the gradients grow exponentially.
- Typical feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.
- In an RNN trained on long sequences (e.g. 100 time steps) the gradients can easily explode or vanish.
  1. We can avoid this by initializing the weights very carefully.
- Even with good initial weights, its very hard to detect that the current target output depends on an input from many time-steps ago.

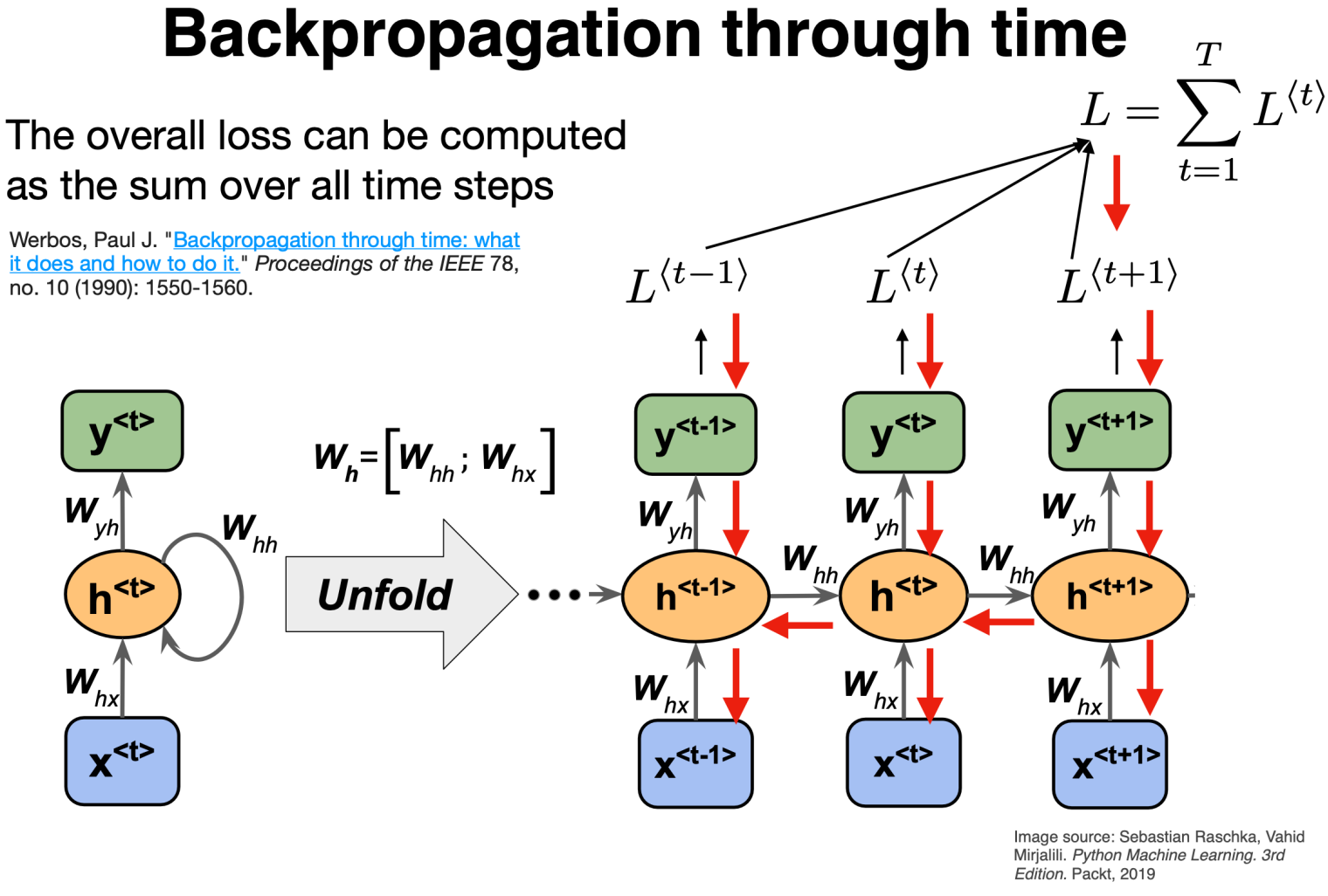
RNNs have difficulty dealing with long-range dependencies.

Mathematical setup

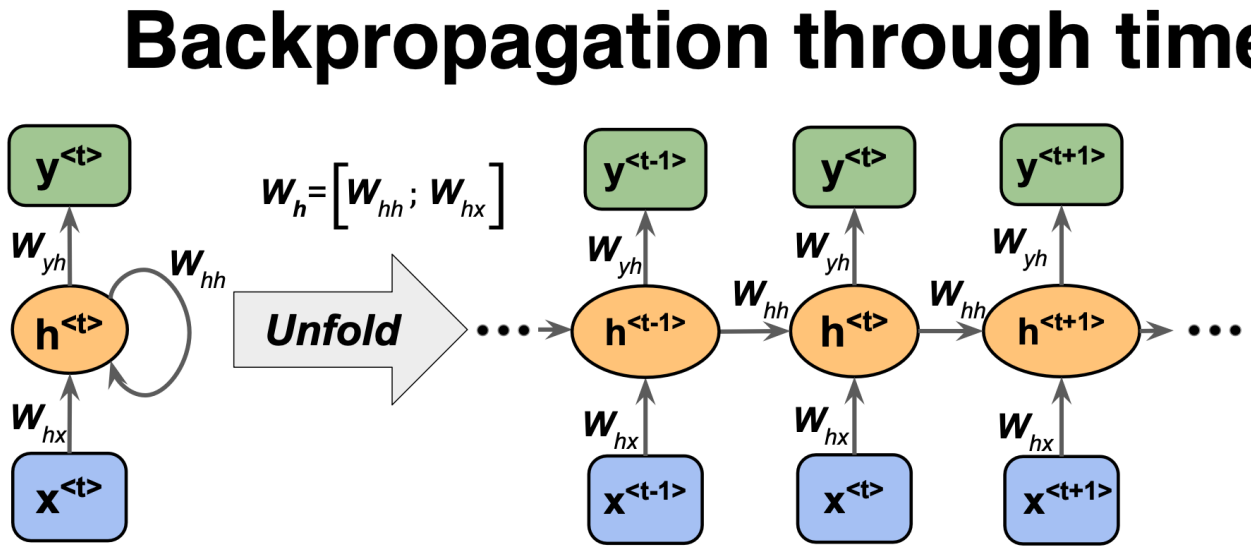
The expression for the simplest Recurrent network resembles that of a regular feed-forward neural network, but now with the concept of temporal dependencies

$$
\begin{aligned}
\mathsf{a}^{(t)} &= U * \mathsf{x}^{(t)} + W * \mathsf{h}^{(t-1)} + \mathsf{b}, \\
\mathsf{h}^{(t)} &= \sigma_h(\mathsf{a}^{(t)}), \\
\mathsf{y}^{(t)} &= V * \mathsf{h}^{(t)} + \mathsf{c}, \\
\hat{\mathsf{y}}^{(t)} &= \sigma_y(\mathsf{y}^{(t)}).
\end{aligned}
$$

Back propagation in time through figures, part 1



Back propagation in time, part 2



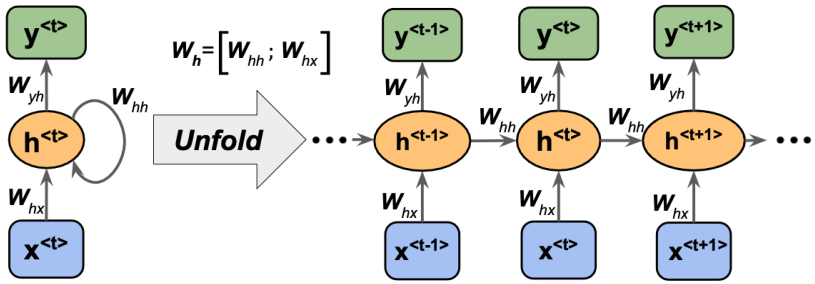
Werbos, Paul J. "Backpropagation through time: what it does and how to do it." *Proceedings of the IEEE* 78, no. 10 (1990): 1550-1560.

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

$$\frac{\partial L^{(t)}}{\partial \mathbf{W}_{hh}} = \frac{\partial L^{(t)}}{\partial y^{(t)}} \cdot \frac{\partial y^{(t)}}{\partial \mathbf{h}^{(t)}} \cdot \left(\sum_{k=1}^{t} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} \cdot \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}_{hh}} \right)$$

Back propagation in time, part 3

Backpropagation through time



Werbos, Paul J. "Backpropagation through time: what it does and how to do it." *Proceedings of the IEEE* 78, no. 10 (1990): 1550-1560.

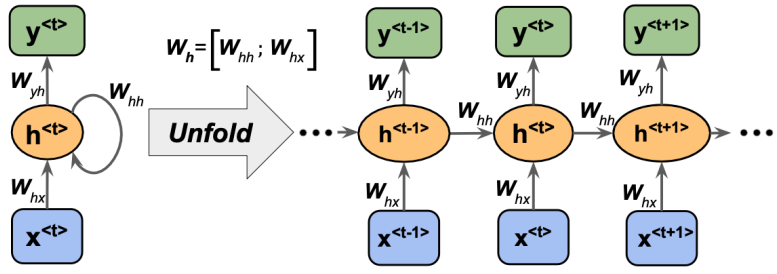
$$\frac{\partial L^{(t)}}{\partial \mathbf{W}_{hh}} = \frac{\partial L^{(t)}}{\partial y^{(t)}} \cdot \frac{\partial y^{(t)}}{\partial \mathbf{h}^{(t)}} \cdot \left(\sum_{k=1}^{t} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} \cdot \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}_{hh}} \right)$$

computed as a multiplication of adjacent time steps:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}}$$

Back propagation in time, part 4

Backpropagation through time



Werbos, Paul J. "Backpropagation through time: what it does and how to do it." *Proceedings of the IEEE* 78, no. 10 (1990): 1550-1560.

$$L = \sum_{t=1}^{T} L^{(t)} \qquad \frac{\partial L^{(t)}}{\partial \mathbf{W}_{hh}} = \frac{\partial L^{(t)}}{\partial y^{(t)}} \cdot \frac{\partial y^{(t)}}{\partial \mathbf{h}^{(t)}} \cdot \left(\sum_{k=1}^{t} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} \cdot \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}_{hh}} \right)$$

computed as a multiplication of adjacent time steps:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}}$$

This is very problematic:
Vanishing/Exploding gradient problem!

Back propagation in time in equations

To derive the expression of the gradients of $\mathcal{L}$ for the RNN, we need to start recursively from the nodes closer to the output layer in the temporal unrolling scheme - such as y and h at final time $t = \tau$,

$$(\nabla_{\mathsf{y}^{(t)}} \mathcal{L})_i = \frac{\partial \mathcal{L}}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial y_i^{(t)}},$$

$$\nabla_{\mathsf{h}^{(\tau)}} \mathcal{L} = \mathsf{V}^{\mathsf{T}} \nabla_{\mathsf{y}^{(\tau)}} \mathcal{L}.$$

Chain rule again

For the following hidden nodes, we have to iterate through time, so by the chain rule,

$$\nabla_{\mathsf{h}^{(t)}} \mathcal{L} = \left(\frac{\partial \mathsf{h}^{(t+1)}}{\partial \mathsf{h}^{(t)}} \right)^{\mathsf{T}} \nabla_{\mathsf{h}^{(t+1)}} \mathcal{L} + \left(\frac{\partial \mathsf{y}^{(t)}}{\partial \mathsf{h}^{(t)}} \right)^{\mathsf{T}} \nabla_{\mathsf{y}^{(t)}} \mathcal{L}.$$

Gradients of loss functions

Similarly, the gradients of $\mathcal{L}$ with respect to the weights and biases follow,

$$\nabla_{\mathsf{c}}\mathcal{L} = \sum_t \left(\frac{\partial \mathsf{y}^{(t)}}{\partial \mathsf{c}}\right)^{\mathsf{T}} \nabla_{\mathsf{y}^{(t)}}\mathcal{L}$$

$$\nabla_{\mathsf{b}}\mathcal{L} = \sum_t \left(\frac{\partial \mathsf{h}^{(t)}}{\partial \mathsf{b}}\right)^{\mathsf{T}} \nabla_{\mathsf{h}^{(t)}}\mathcal{L}$$

$$\nabla_{\mathsf{V}}\mathcal{L} = \sum_t \sum_i \left(\frac{\partial \mathcal{L}}{\partial y_i^{(t)}}\right) \nabla_{\mathsf{V}^{(t)}} y_i^{(t)}$$

$$\nabla_{\mathsf{W}}\mathcal{L} = \sum_t \sum_i \left(\frac{\partial \mathcal{L}}{\partial h_i^{(t)}}\right) \nabla_{\mathsf{w}^{(t)}} h_i^{(t)}$$

$$\nabla_{\mathsf{U}}\mathcal{L} = \sum_t \sum_i \left(\frac{\partial \mathcal{L}}{\partial h_i^{(t)}}\right) \nabla_{\mathsf{U}^{(t)}} h_i^{(t)}.$$

Summary of RNNs

Recurrent neural networks (RNNs) have in general no probabilistic component in a model. With a given fixed input and target from data, the RNNs learn the intermediate association between various layers. The inputs, outputs, and internal representation (hidden states) are all real-valued vectors.
In a traditional NN, it is assumed that every input is independent of each other. But with sequential data, the input at a given stage t depends on the input from the previous stage $t-1$

Summary of a typical RNN

1. Weight matrices U, W and V that connect the input layer at a stage t with the hidden layer h_t, the previous hidden layer h_{t-1} with h_t and the hidden layer h_t connecting with the output layer at the same stage and producing an output $\tilde{y}_t$, respectively.
2. The output from the hidden layer h_t is oftem modulated by a tanh function $h_t = \sigma_h(x_t, h_{t-1}) = \tanh(Ux_t + Wh_{t-1} + b)$ with b a bias value
3. The output from the hidden layer produces $\tilde{y}_t = \sigma_y(Vh_t + c)$ where c is a new bias parameter.
4. The output from the training at a given stage is in turn compared with the observation y_t thorugh a chosen cost function.

The function g can any of the standard activation functions, that is a Sigmoid, a Softmax, a ReLU and other. The parameters are trained through the so-called back-propagation through time (BPTT) algorithm.

Four effective ways to learn an RNN and preparing for next week

1. Long Short Term Memory Make the RNN out of little modules that are designed to remember values for a long time.
2. Hessian Free Optimization: Deal with the vanishing gradients problem by using a fancy optimizer that can detect directions with a tiny gradient but even smaller curvature.
3. Echo State Networks: Initialize the input a hidden and hidden-hidden and output-hidden connections very carefully so that the hidden state has a huge reservoir of weakly coupled oscillators which can be selectively driven by the input.
   - ESNs only need to learn the hidden-output connections.
4. Good initialization with momentum Initialize like in Echo State Networks, but then learn all of the connections using momentum

Long Short Term Memory (LSTM)

LSTM uses a memory cell for modeling long-range dependencies and avoid vanishing gradient problems.

1. Introduced by Hochreiter and Schmidhuber (1997) who solved the problem of getting an RNN to remember things for a long time (like hundreds of time steps).
2. They designed a memory cell using logistic and linear units with multiplicative interactions.
3. Information gets into the cell whenever its "write" gate is on.
4. The information stays in the cell so long as its **keep** gate is on.
5. Information can be read from the cell by turning on its **read** gate.