Models for sequential data (RNN)

Deep Learning (2IMM10)
Spring 2020

5 Models for sequential data (RNN)

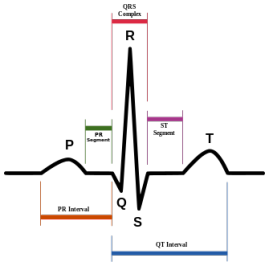
The learning outcomes of this chapter:

- Able to develop models for sequential data using recurrent neural networks

6 Temporal correlations

In this chapter we study models for tasks where the data is typically sequential with long distance correlations. Let us examine the following example:

We have data collected over a period of time from senor measurements of the electrical activity of the human hearth. These measurements form an Electro Cardiogram, or ECG, depicted in Figure 1.



(a) The QRS complex observed in an ECG.

(b) Multiple QRS complices.

Figure 1: Electro cardiogram (ECG)

The QRS complex observed in the ECG, consist of the Q wave, the R wave and the S wave — see Figure 1a. Without going into any medical details as this is not the purpose of the example, we only define our task as detecting the length of the complex as this is relevant diagnostic information.

To be able to estimate the length of the QRS, basically we need to detect the individual components and measure the time from the beginning to the end of the complex.

As the Q-wave, R-wave and S-wave have a particular shape, one can safely assume that a CNN model can develop detectors for those shapes — see Figure 2 for this.

When dealing with sequential data, we often encounter very long signals, particularly during execution of the model. In general we can often assume that our model does not need to see the whole signal to produce the output. In this particular case, we use a sliding window approach, where our

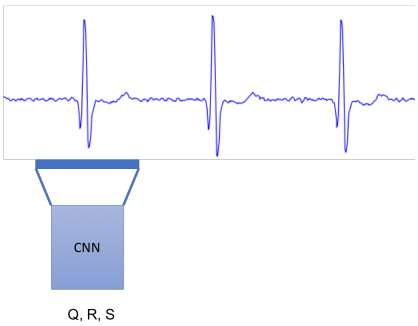
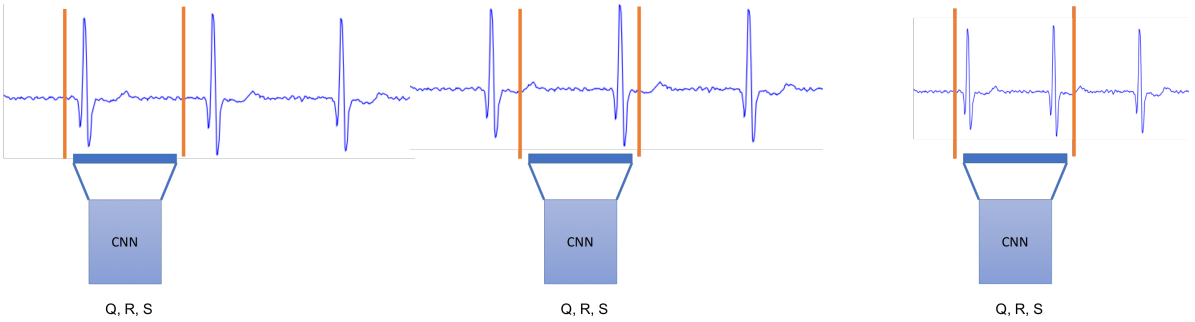


Figure 2: CNN model for detecting the QRS complex



(a) The QRS complex located at the beginning of the window

(b) The QRS Object at the end of the window

(c) Two QRS objects within the window

Figure 3: The QRS complex can be present in various spots of the sliding window, or even at multiple points in the same window.

model looks at a window of data that is slid over the data — see Figure 3a. Detecting the complex with such a CNN model involves a number of decisions, one of which is the size of the window.

The task for our model is to estimate the length of the QRS complex. To be able to achieve this using a sliding window approach, the window size must be able to fit the maximum length of QRS complex that we expect to encounter (or aim to detect).

As that would be the upper limit, most of the QRS complexes in the signal will have a shorter length. This means we can expect the QRS patterns to appear in different locations in the window — see Figures 3b and 3c.

We can also expect that for a select range of QRS-length and frequency of appearance of the pattern, no QRS-objects may land in a window — see fig. 4.

Because we have so many different cases, when preparing the training data we have to make sure that all of them are represented. Assuming that our original training dataset consist of long sequences, first we need to split the data into window length images add specify a label for each image. We expect that the labels are difficult to compute automatically with rule based algorithm,[1] so we rely on an expert to provide the labels. To achieve good generalization, our dataset should

[1]Otherwise there's really no need to train an ML model.

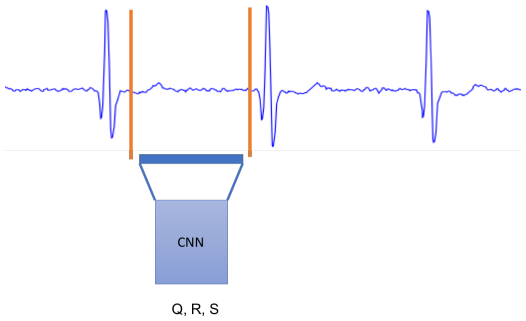


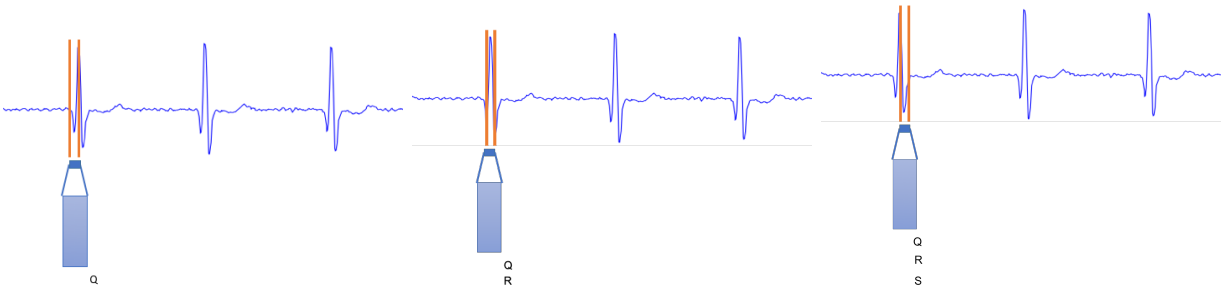
Figure 4: A window containing no QRS-complexes.

have a good coverage of cases with different lengths — from a typically shortest to the longest QRS complex. However, as we also expect that the object can appear at every location in the image, this adds another degree of freedom to the range of variations that need to be covered with training data. In other words, our training data should contain different length examples should in different locations of the window.

This need to cover all possible cases in the training data can become a significant drawback. Particularly in task where the correlations need to be uncovered from events that happen over longer distances. This means we need a large window size, that in turn adds computational complexity and increases the number of parameters significantly in the dense layers. Moreover, the need to cover all cases can mean we need to create a very large training set which can be laborious and costly.

6.1 A different type of model

In contrast, imagine a different type of model. One that looks at a very small window, detects the Q-wave. Then takes a number of steps until it reaches the R-wave. It then detects that one, moves forward until it reaches the S-wave and detects this last part of the pattern — see Figure 5. Can this model perform the same task that the CNN model did? What would this model need to be able to achieve the same task?



(a) The model first detects the Q-wave.

(b) It then continues to detect the R-wave.

(c) Finally it detects the S-wave.

Figure 5: A different type of model for sequential data.

Basically for such a model to detect the length of the QRS complex, it needs to be able to perform

the following tasks:

- Detect Q, and R, and S patterns
- Remember the the point of Q
- Remember the point of R
- Remember the point of S
- Compute the distance from Q to R to S

One requirement that is not met by the CNN is that this model needs *memory*. As we have seen so far all feed-forward models process the input directly to produce output and are not capable of storing information in between consecutive inputs. To meet these requirements we introduce the recurrent neural network (RNN).

7 Recurrent neural networks

7.1 Introduction

One type of network that can learn to perform the tasks above, is the *Recurrent Neural Network* (RNN). This type of network can

- process a sequence of datapoints one at a time;
- produce an output at each step[2];
- produce a memory state at each step;
- combine information from datapoints at different times to compute an output.

To achieve these feature the RNN model introduces a internal 'hidden state' and links with a delay, so that the hidden state at the end of one time step is used to do computations in the next — see Figure 6a.

On a high level the model now has the following components

- the input x_t
- the hidden state h_t
- the output y_t

each of these is indexed with the time step t. Here h_t is defined as a function of both x_t and h_{t-1}, that is

$$h_t = f_h(x_t, h_{t-1}),$$

and y is defined as in the MLP as a function of the hidden state of the model, so

$$y_t = f_y(h_t).$$

You can think of the connection between h_t and h_{t-1} as a delay node, as indicated in Figure 6a.

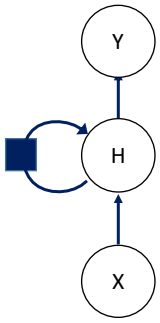
Another way that we look at the RNN models is in an unrolled form — see Figure 6b. This form is useful to look at the computations during training in detail. Note that here the edges represent the maps

$$f_h : x_t,\, h_{t-1} \mapsto h_t,$$
$$f_y : h_t \mapsto y_t.$$

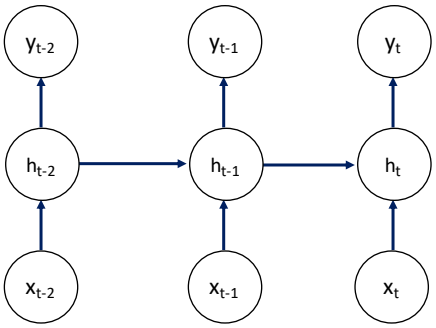
These maps themselves do not depend on t, which means that the model reuses the parameters[3] for each time step.

[2] Although it doesn't have to.

[3] To prevent cluttering of the notation we don't include the parameters as subscript, but of course, as in the previous chapters, these functions are parameterized and the goal will be to find the right parameters again.



(a) The square in the connection from the hidden state to itself indicates a delay.



(b) We can display the same architecture in an "unrolled" way.

Figure 6: A schematic overview of an RNN.

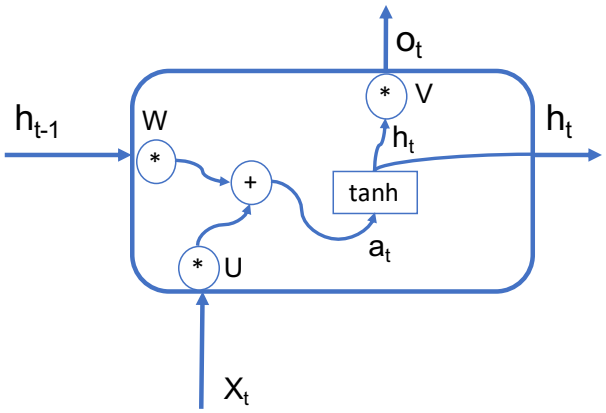


Figure 7: Vanilla RNN

We implement the conceptual model given in Figure 6a as follows:

$$h_t = \phi_1(Wh_{t-1} + Ux_t + b),$$
$$y_t = \phi_2(Vh_t + c),$$

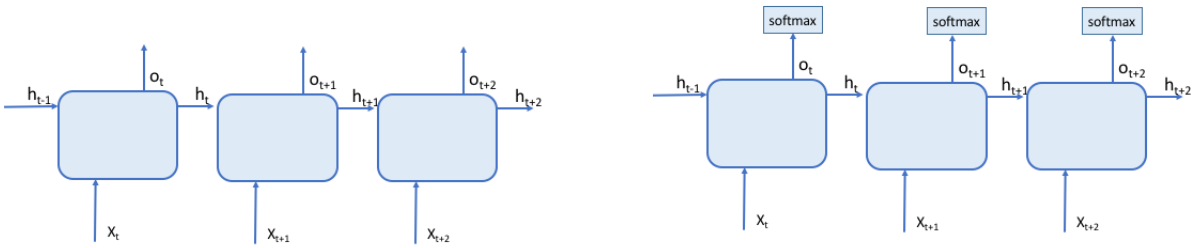
where W, U, b, V, c are parameters of the model, and ϕ_1 and ϕ_2 are activation functions. Figure 7 depicts the implementation diagram of the model, which we specifically refer to as the RNN cell.

If we use $\phi_1 = \tanh$ and $\phi_2 = \mathrm{id}$, we get the cell shown in Figure 7, where

$$\left.\begin{aligned} a_t &= Wh_{t-1} + Ux_t + b, \\ h_t &= \tanh(a_t), \\ o_t &= Vh_t + c. \end{aligned}\right\} \tag{1}$$

Figure 8a depicts the unrolling of this model.

Let us look at some tasks for which we can use an RNN. For a task of sequence classification, we have the following training data: $D : \{(x_i)_0^{N-1}, (y_i)_0^{N-1}\}$. The the input sequence x is of length N and the target variable y is a assigned a value (label) from a discrete set of values $y \in S$.



(a) Unrolled RNN

(b) Unrolled RNN for sequence classification.

Figure 8: The RNN unrolled.

We can the use the softmax output function to model a discrete probability distribution over the set of classes in S, like in Figure 8b. This means that on top of eq. (1), we have

$$y^{(t)} = \text{softmax}(o^{(t)})$$

for the output.

A simple example of such a task would be to take the running sum of a sequence of integers modulo 10. In this case $(x_i)_0^{N-1}$ is a sequence of integers and $(y_i)_0^{N-1}$ is a sequence of elements from $S = \{0, \ldots, 9\}$.

Note: This task is a great example of a task that is not well motivated for a Machine Learning solution. It is no surprise that a ML model will do a substandard job at this task compared to any calculator as the calculator can easily achieve full accuracy.

In fig. 8b we can see this model unrolled. This model implements the conditional probability distribution over S given in eq. (2).

$$P(y_t|x_t, x_{t-1}, ..., x_0) \tag{2}$$

The output of such models is determined by selecting the most likely value for y_t at time t as eq. (3)

$$\arg\max_{s_t \in S} P(y_t = s_t|x_t, x_{t-1}, ..., x_0) \tag{3}$$

where s_t are is an element of $s_t \in S$.

In this case we select the value for y_t independently of other values y_t as shown in eq. (4).

$$P(y_0, ..y_N) = P(y_0)P(y_1)...P(y_N) \tag{4}$$

Architectures for other tasks

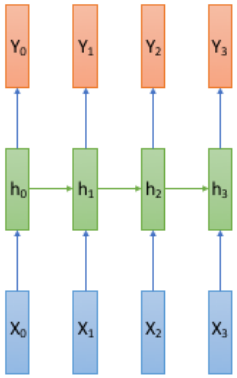
We give a schematic representation of the architecture we have discussed so far in Figure 9a. Other commonly used architectures with an RNN cell are depicted in Figures 9b to 9d.

The architecture shown in Figure 9b implements the model given in eq. (5), where we want to determine what class the entire sequence belongs to. In this case we have a single label y for the sequence of input $(x_t)_0^N$. We implement this by ignoring all except the last output of the RNN cell.

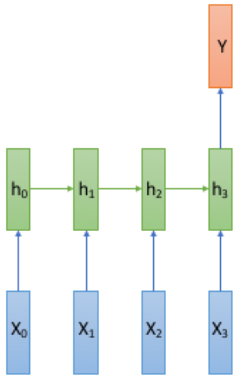
$$P(y|x_N, x_{N-1}, ..., x_0) \tag{5}$$

In contrast to this the architecture given in Figure 9c takes a single input x and produces a sequence of outputs $(y_t)_0^N$. An example of a task for which this model is well suited is the captioning of images. To implement this we can feed the (processed) image to the RNN in the first step, and in subsequent steps we can feed it an empty image or we can just feed it the same image in each time step.

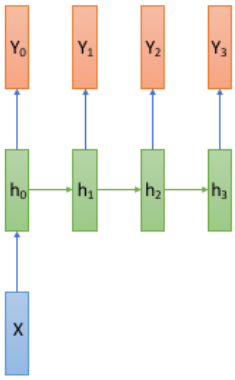
Another set of tasks fit into the description of sequence-to-sequence mapping. In these cases both the input and the output are sequences, but they are of different length. A solution using a single RNN cell is depicted in Figure 9d. One example for such a task is translation of a sentence from one language to another. Typically for such tasks a more complex architecture with multiple components (including more than one RNN cell) is better suited. We will discuss these sequence-to-sequence models in more detail later in this chapter.



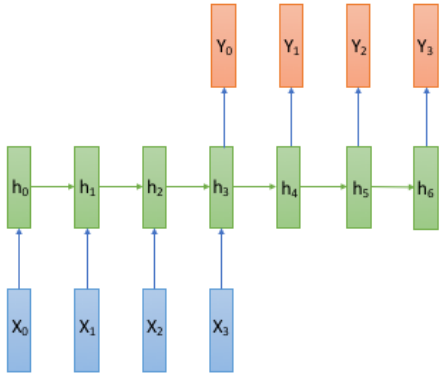
(a) RNN used for aligned sequence classification



(b) RNN used for sequence classification

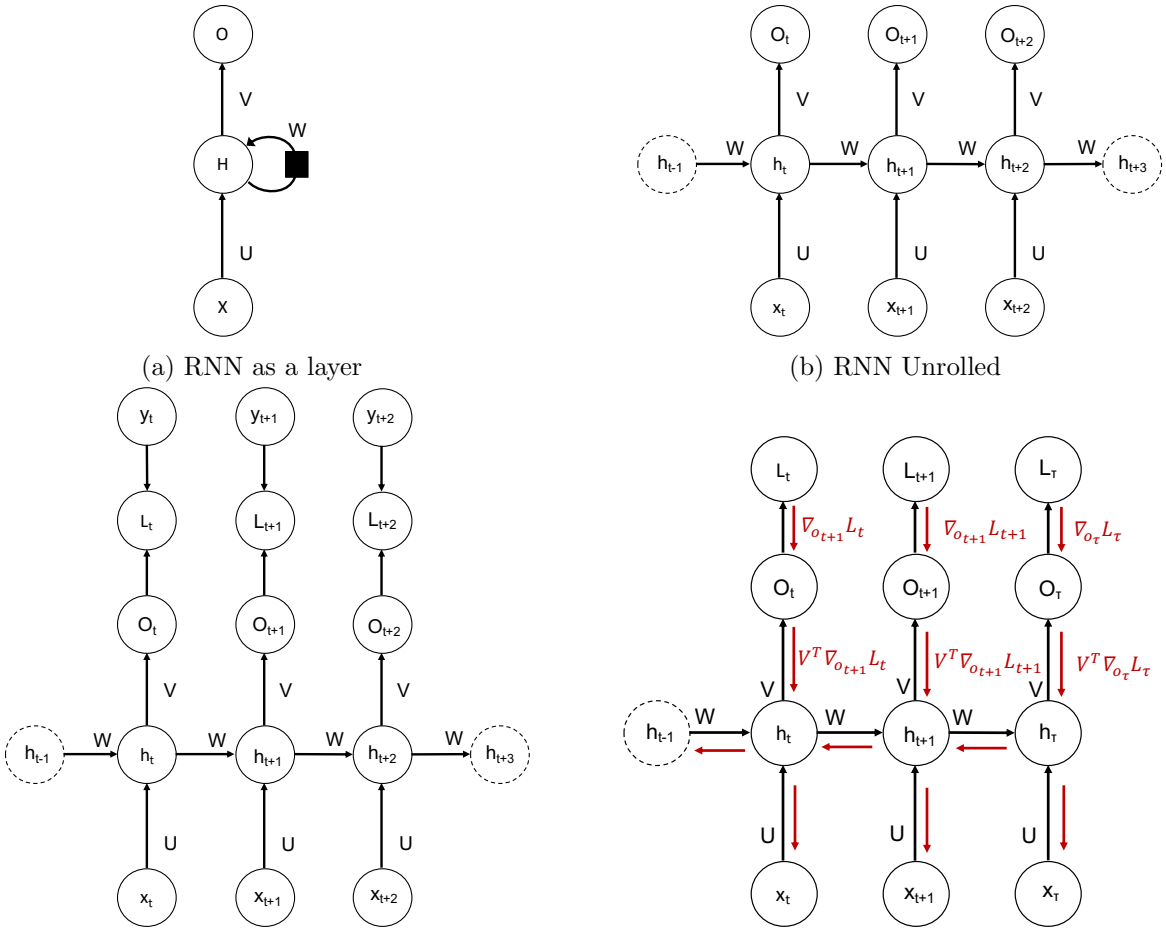


(c) RNN used for sequence generation



(d) RNN used for the sequence to sequence model

Figure 9: RNN architectures for various common types of tasks.



(a) RNN as a layer

(b) RNN Unrolled

(c) RNN including loss function unrolled.

(d) Backpropagation through time[4]

Figure 10: The training of an RNN unrolled.

7.2 Backpropagation through time

As with the other neural network models we have discussed so far, we use SGD to train the parameters of the RNN models. However, backprogatation through the delay node of the RNN cell cannot happen directly — from Figure 10a there is no obvious way to calculate the gradients directly. Rather, training of RNN models first requires that the model is unrolled as in Figure 10b. This variation of backpropagation is called *backpropagation through time*.

For simplicity, let us first look at the case where we only have a loss at the final time step, i.e.

$$L = L_\tau.$$

We will walk through this process for the RNN cell described in eq. (1).

We start the backpropagation from the end of the model as usual, and we go backward to the beginning.

[4]N.B. for a function $f : \mathbb{R}^n \to \mathbb{R} : \mathbf{x} \mapsto f(\mathbf{x})$ the gradient is the transposed of the total derivative, i.e. $\nabla_{\mathbf{x}} f = J_f(\mathbf{x})^\top = \left(\frac{\partial f}{\partial \mathbf{x}}\right)^\top$.

Then we take the gradient of each signal with respect to previous signals as given in eq. (7).

$$\begin{aligned}
\frac{\partial L}{\partial h_\tau} &= \frac{\partial L_\tau}{\partial O_\tau} V \\
\frac{\partial L}{\partial h_t} &= \frac{\partial L}{\partial h_\tau}\frac{\partial h_\tau}{\partial h_t} && \text{for all } t < \tau \\
\frac{\partial h_\tau}{\partial h_{\tau-1}} &= \operatorname{diag}(\phi'(W h_{\tau-1} + U x_\tau + b))W \\
\frac{\partial h_\tau}{\partial h_t} &= \operatorname{diag}(\phi'(W h_{\tau-1} + U x_\tau + b))W \cdots \operatorname{diag}(\phi'(W h_t + U x_{t+1} + b))W && \text{for all } t < \tau \quad (6) \\
\frac{\partial h_\tau}{\partial W} &= \frac{\partial h_\tau}{\partial W^{(\tau)}} + \frac{\partial h_\tau}{\partial h_{\tau-1}}\frac{\partial h_{\tau-1}}{\partial W} && (7)
\end{aligned}$$

Here by $\frac{\partial h_\tau}{\partial W^{(\tau)}}$ we mean the derivative we get when we view the W in every time step separately.[5]This becomes

$$\begin{aligned}
\frac{\partial h_\tau}{\partial W_{ij}^{(\tau)}} &= \operatorname{diag}(\phi'(W h_{\tau-1} + U x_\tau + b))(h_{\tau-1,j}\, \mathbf{e}^i), \\
&= \phi'(W h_{\tau-1} + U x_\tau + b)_i \cdot h_{\tau-1,j}\, \mathbf{e}^i,
\end{aligned}$$

where $\mathbf{e}^i$ is the vector with all zeros except at index i where it has a one.

In contrast to the feedforward networks, we see in that in the unrolled RNN, depicted in Figure 10d, that the parameters are shared at each time step. Therefore, the parameters will receive updates proportional to the gradients from the current time step and all future time steps.

To illustrate this let us look at how W is updated. For the model we are looking at, the derivative of the loss with respect to W is computed as

$$\begin{aligned}
\frac{\partial L}{\partial W} &= \frac{\partial L}{\partial h_\tau}\frac{\partial h_\tau}{\partial W} \\
&= \frac{\partial L}{\partial h_\tau}\left(\frac{\partial h_\tau}{\partial W^{(\tau)}} + \frac{\partial h_\tau}{\partial h_{\tau-1}}\frac{\partial h_{\tau-1}}{\partial W}\right) \\
&= \frac{\partial L}{\partial h_\tau}\sum_t \frac{\partial h_\tau}{\partial h_t}\frac{\partial h_t}{\partial W^{(t)}}.
\end{aligned}$$

If instead of only having a loss at the final timne step, we have a loss based on the output at every time step, i.e.

$$L = \sum_t L_t,$$

the gradient of the loss with respect to W becomes a sum over the gradients with respect to all of

[5]What we mean here is that we can see h_τ a function of the form $h_\tau(W, h_{\tau-1}(W))$ and compute the derivative wrt W as $\frac{\partial h_\tau(W, h_{\tau-1}(W))}{\partial W} = \frac{\partial h_\tau(A, h_{\tau-1}(B))}{\partial A} + \frac{\partial h_\tau(A, h_{\tau-1}(B))}{\partial B}$ where we set A and B to W. So with $\frac{\partial h_\tau}{\partial W^{(\tau)}}$ we mean $\frac{\partial h_\tau(A, h_{\tau-1}(B))}{\partial A}$ with A and B set to W.

those losses. That means we get

$$\begin{aligned}\frac{\partial L}{\partial W} &= \sum_{t\leq\tau} \frac{\partial L_t}{\partial W} \\ &= \sum_{t\leq\tau} \frac{\partial L_t}{\partial h_t} \sum_{s\leq t} \frac{\partial h_t}{\partial h_s} \frac{\partial h_s}{\partial W^{(s)}}. \end{aligned} \tag{8}$$

If we rearrange these terms, we can see how updates from future losses determine how the weights should be changed at a certain time step:

$$(8) = \sum_{s\leq\tau} \left(\sum_{t=s}^{\tau} \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_s} \right) \frac{\partial h_s}{\partial W^{(s)}}. \tag{9}$$

Note that, as shown in eq. (6), $\frac{\partial h_t}{\partial h_s}$ is a product of $t-s$ terms, so the further in the future the loss is, the harder it becomes to get updates from it due to vanishing (or exploding) gradients.

To make this a bit clearer, let us look at the RNN unrolled, and let us say that instead of having the same weights W at every time step, we have different weight matrices $W^{(t)}$ at different times t. The term

$$\frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_s} \frac{\partial h_s}{\partial W^{(s)}}$$

then tells us how the weights $W^{(s)}$ at time s should be updated to decrease the loss L_t at time t. If all the matrices in the product $\frac{\partial h_t}{\partial h_s}$ have norm less than 1, this product will be smaller the more factors it has,[6]i.e. the further t and s are apart, and so the less effect the loss at time t will have on how we update the weights at time s.

The difference when we do share the parameters between time steps is that to get the gradient with respect to W we sum over all the gradients with respect to $W^{(s)}$, as shown in eq. (9). But since for every s we get very little information from losses far from s, this means our RNN has a hard time learning long range dependencies.

To recap, we can draw two conclusions. As the model is feeding forward to the update as well as to future steps, we get update values both from the loss at the current step as well as from all future losses. Secondly, as the parameters (in our example W) are re-used at each time step the full update for them is the sum over all time steps — see eq. (9).

This realization has important consequences. For one, we need to think differently about depth when using RNN models. Not only is depth in these models proportional to the number of layers, but it is also proportional to the number of steps in the input sequence. This implies that the same challenges we face when dealing with deeper networks — such as vanishing gradients — also come up when we try to use RNNs to model long sequences. Note that, in the computation of the gradients during backpropagation, the parameters of the model are a factor of a product with a length of up to the length of the sequence. This can lead to exploding or vanishing gradients. This limits the ability of this 'vanilla' version of the RNN cell to be used on longer sequences and to model longer term dependencies[7].

[6]Note that if ϕ is the hyperbolic tangent function, its derivative ϕ' is less than, or equal to 1, with equality only in 0.

[7]Bengio, Yoshua, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult." IEEE transactions on neural networks 5.2 (1994): 157-166.

Different innovations in the RNN cell have been developed to add address these difficulties. Most notably the introduction of the Long short-term memory (LSTM) model[8]. This model introduces a protected cell that allows for gradients to flow freely across the different time steps. Before we go into the details of the LSTM we go over the gating mechanism that allows for developing protected cells.

7.3 Gating Mechanism

In the vanilla RNN we have discussed so far, the hidden state is computed as

$$\begin{aligned} a^{(t)} &= Wh^{(t-1)} + Ux^{(t)} + b \\ h^{(t)} &= \tanh(a^{(t)}). \end{aligned}$$

The hidden state at time t are directly affected by its values at the previous time, $t-1$, and the current input x_t. As such, to be able to store information in h needed to produce a specific output at a later time, the parameters in W need to be very well-tuned. They are responsible for making sure that information will be stored for a given input (and previous hidden values) as well as protected once stored until it is needed to produce the specific output. Finding such parameters during training is challenging. (You can read about the limitations in a theoretical study by Benio et al. 1994).

One of the main ways to overcome this problem is to *protect the state of the RNN*. That is, rather than updating the state with each datapoint, we learn

- when to update, given the input and the previous hidden state,
- what to update given the input and the previous state,
- even more so, what to remove (forget),
- and what to add into the memory

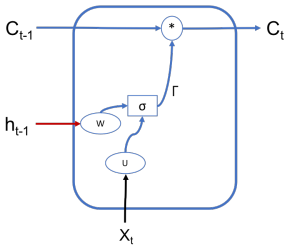
separately. We do this using "gates".

A gate is a mechanism that allows the model to select which information passes through and which is removed. In Figure 11a we introduce a memory cell C into the model. We can observe here how the information in C is controlled based on the values of the hidden state h and the input x — see eq. (10). Specifically, the multiplicative[9] gate in Figure 11a will let information pass if the sigmoidal activations are 1 and will remove information if the activations are 0. Therefore, the parameters in this gate, W_Γ and U_Γ, determine how the hidden state and input remove information from the memory cell C.

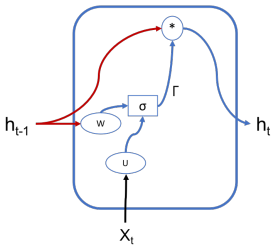
$$\left.\begin{aligned} \Gamma &= \sigma(W_\Gamma h_{t-1} + U_\Gamma x_t + b) \\ C_t &= \Gamma \cdot C_{t-1} \end{aligned}\right\} \tag{10}$$

[8] Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.

[9] The multiplication is performed element-wise.



(a) We can use a gate to remove information from the memory cell, C. See eq. (10) for details.



(b) We can also apply a gating mechanism to the hidden state directly. See eq. (11) for details.

Figure 11: Gating mechanisms

Such mechanism can be even applied to the hidden state itself without introducing a memory cell as shown in eq. (11) and Figure 11b.

$$\left.\begin{aligned} \Gamma &= \sigma(W_\Gamma h_{t-1} + U_\Gamma x_t + b) \\ h_t &= \Gamma \cdot h_{t-1} \end{aligned}\right\} \tag{11}$$

Throughout this chapter, σ denotes a logistic sigmoid function.

7.4 Long short-term memory

The Long short-term memory, or LSTM, architecture introduces a memory cell and a number of gates to control the access to this cell and to how the output is produced. The memory cell can flow directly from one step to another, see Figure 12a. Its value is affected by the forget gate and the add gate.

The forget gate multiplies the cell values element-wise with vector values in the range $[0, 1]$. The values of the gate are specified by the hidden state h_{t-1} and the input x_t as in eq. (12).

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{12}$$

The add gate determines the information that is added to the cell state. The LSTM cell determines the candidate value C'_t based on the values of the hidden state h_{t-1} and the input x_t as shown in eq. (13). These candidate values are passed through the add gate as in eq. (14).

$$C'_t = \tanh(W_C[h_{t-1}, x_t] + b_C) \tag{13}$$

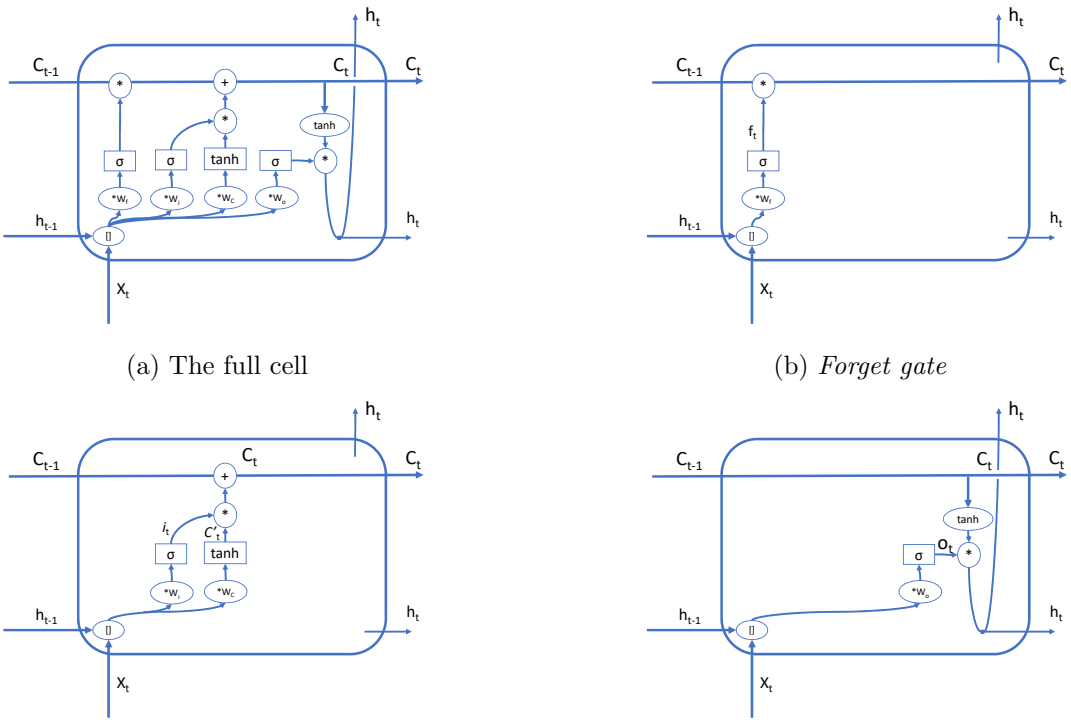
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{14}$$

This makes that the new cell state becomes

$$C_t = f_t \cdot C_{t-1} + i_t \cdot C'_t.$$

The output of the LSTM is computed based on the cell state, however, it also gated by the values of the hidden state and the input as shown in Figure 12d and eq. (15).

$$\left.\begin{aligned} o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) \\ h_t &= o_t \cdot \tanh(C_t) \end{aligned}\right\} \tag{15}$$



(a) The full cell

(b) *Forget gate*

(c) *Add gate*

(d) *Output gate*

Figure 12: The LSTM cell with its gates.

In summary in the LSTM RNN cell, in contrast to the 'vanilla' RNN cell, the hidden state and the input control the flow of information in and out of the cell rather than carrying the information that the model need to produce the output. Analogously when training this model the updates coming from the gradients of the loss can flow back through the cell values. This in turn allows for training parameters of all the gates such that the model can learn long term dependencies in the data much more efficiently. Note that in this LSTM architecture, the hidden state h_t is used as the output of the cell.

7.5 Gated Recurrent Unit

Since its introduction there have been many variations proposed on the LSTML model (including some by the original authors). Nevertheless, the LSTM model still maintained its good standing when it comes to perfomance. In this subsection we consider one variation of the LSTM model that showed some improvements in performance on certain datasets. This, however, is not the motivation to introduce this model, but rather to illustrate how the hidden state can take the role of the cell, while the gating mechanisms still provide the advantage of being able to capture long term dependence. We briefly mentioned before that the gate can be applied to the hidden state. This approach is introduced in the gated recurrent unit (GRU) model shown in Figure 13. In that sense, the GRU model is a simplificaiton of the LSTM and has fewer parameters.

The GRU has two gates: a "reset gate" r and an "update gate" z. The update gate determines what information in the hidden state should be updated, and the reset gate determines what information

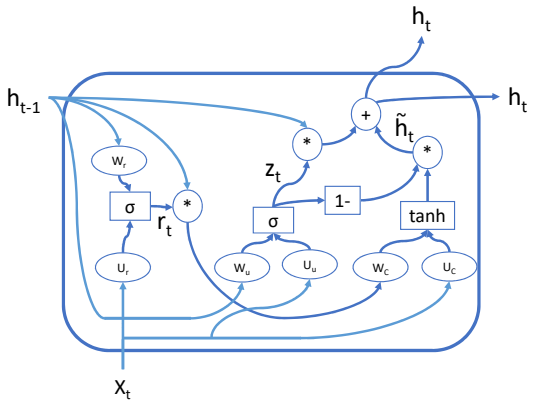


Figure 13: Gated Recurrent Unit

in the old hidden state should be ignored when computing the update. The gates are computed as

$$z_t = \sigma(W_u h_{t-1} + U_u x_t + b_u),$$
$$r_t = \sigma(W_r h_{t-1} + U_r x_t + b_r),$$

and the proposed update is computed as

$$\tilde{h}_t = \tanh(W_c(r_t \cdot h_{t-1}) + U_c x_t + b_c).$$

Finally the new value for the hidden state is then

$$h_t = z_t \cdot h_{t-1} + (1 - z_t) \cdot \tilde{h}_t.$$

8 RNN models

We look at a number of models that include RNN cells for different tasks.

8.1 Sequence classification

For a task of sequence classification, where a variable length sequence needs to be classified or regressed to a single value we can often use the model in Figure 14. This model is especially suitable for situations where the input comes from a fixed-size set such as the characters in the alphabet.

The RNN cell can also be used as a component of more complex models depending on the structure of the input data. For example, let us look at the motivating example at the beginning of this chapter again: the QRS complex. There we have both tasks based on localized correlations — detecting the separate Q, R, and S waves — as well as a task based on the longer term correlations of following the sequence of those 3 events and calculating the length of the complex. With this in mind, instead of only using an RNN, as in Figure 15a, we might want to use a combination of convolutional layers and an RNN cell as in Figure 15b.

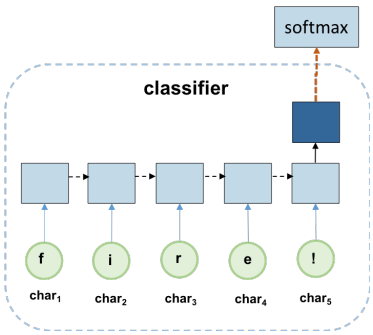
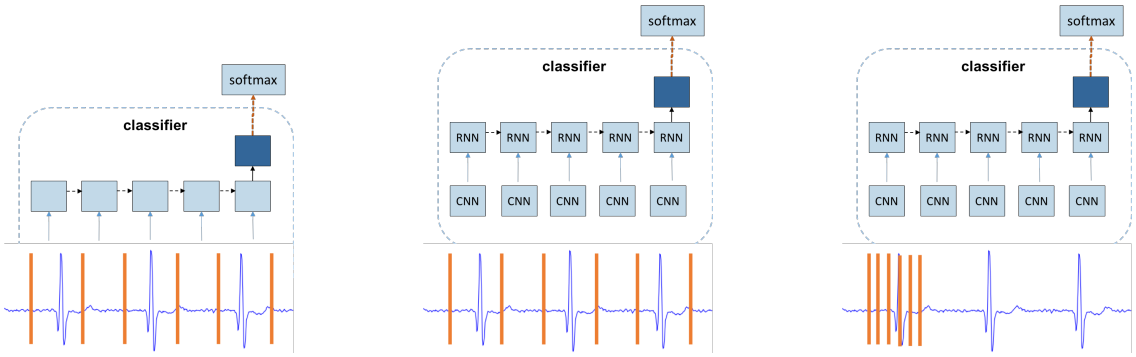


Figure 14: Sequence classifier - Classify variable length sequences



(a) Using just an RNN to measure the length of the QRS-complex.

(b) We can combine the RNN with convolutional layers.

(c) Because the convolutional layers only need to detect highly localized phenomena, the window size can be small.

Figure 15: Models for measuring the length of the QRS-complex.

Note: as we now have an RNN cell to take over the job of detecting the full QRS complex, the responsibility of the convolutional layers is reduced to only detecting the individual waves and hence the convolutional window can be much smaller making the model more efficient. This is depicted in Figure 15c.

Let us look at the problem of sentiment detection in natural text as another example. This is where a short text is classified with a sentiment labels, like positive or negative. Instead of learning representations from scratch, we can use an embedding layer that is pre-trained using the CBOW or Skipgram model. This pre-training does not need labeled data, so we can train the embedding layer on a much larger corpus, especially when training data for the classification task is limited. Alternatively we can make use of a pre-trained embedding layer that is publicly available. Both options are examples of transfer learning. The overall architecture would come down to the one shown in Figure 16.

8.2 Bi-directional RNN

In sequence classification tasks the model processes the whole sequence and produces the class estimation. In such cases it may be useful to process the sequence in both directions before producing

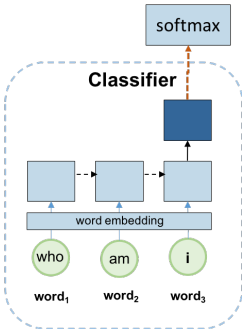
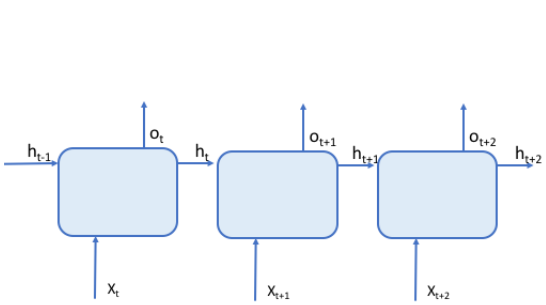
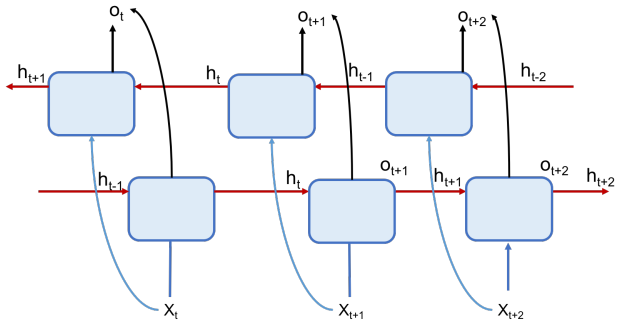


Figure 16: Text classification with RNN and embedding



(a) The RNN we have discussed so far processes the data in one direction.



(b) A bi-directional RNN on the other hand processes the sequence in both directions simultaneously

Figure 17: A regular RNN versus a bi-directional RNN.

the estimation. As a variation of RNN, the bi-directinal RNN[10] actually processes the data in both directions to produce an estimate — see Figures 17a and 17b.

8.3 Sequence to sequence models

So far we have looked at models where either every element of a sequence needs to be classified, or where a sequence needs to be classified as a whole. But sometimes we want to map variable length sequences to other sequences that might not be aligned and have different lengths. Think for example of translating sentences from one language to another. The translated sentences might have very different lengths, and the word order might be completely different. For these kinds of tasks we have sequence to sequence (seq2seq) models.

In general, we would like to have a model of the joint distribution of a sequence $y_0, ...y_n$, given a sequence $x_0, ..., x_k$, as shown in eq. (16). Having such a model allows us to find the most likely sequence $y_0, ...y_n$ for a given input sequence $x_0, ..., x_k$ — i.e. to find eq. (17). From the point of view of the implementation typically there are two components: a model that estimates the joint probability distribution and a search algorithm that produces the sequence that maximizes this probability.

[10]Graves, Alex, Santiago Fernández, and Jürgen Schmidhuber. "Bidirectional LSTM networks for improved phoneme classification and recognition." 2005

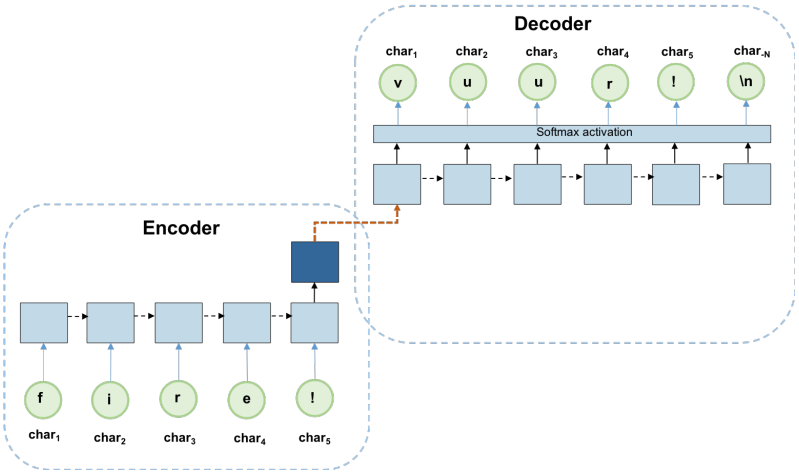


Figure 18: Seq2seq model

$$P(y_0, y_1, ..y_n | x_0, x_1, ...x_k) \tag{16}$$

$$\arg\max_{y_0,...y_n} P(y_0, y_1, ..y_n | x_0, x_1, ...x_k) \tag{17}$$

Figures 9d and 18 depict the conceptual view of a seq2seq model. The seq2seq models are typically implemented with two RNN cells, where one cell acts as an encoder of the input and the other acts as decoder that produces the output.

The encoder's output or its hidden state at the end of the sequence can be used as the *encoding* of the input sequence. This *code* is then used as an initialization of the hidden state of the decoder. In such a configuration the decoder models eq. (18) where h_k is the code.

$$P(y_0, y_1, ..y_n | h_k) \tag{18}$$

Notice the similarities with the autoencoders from Chapter 4 in both terminology and structure. Both models have an encoding and a decoding part. We have a representation of the input produced by the encoder. If we train the seq2seq model with the same input as the output it does take on the role of a sequence autoencoder and can be used for representation learning.

There are variations on the seq2seq model changing e.g. what is used as the code and how the decoder is initialized (conditioned) by the code. There may be specific reasons to use such a variation for some applications, but here we limit the discussion to the given setup.

As the goal of our task is to maximize the joint probability of the output sequence, eq. (17), the model configuration given in Figure 18 actually assumes that the next output symbol is independent of the previously emitted symbols in the sequence, see eq. (19).

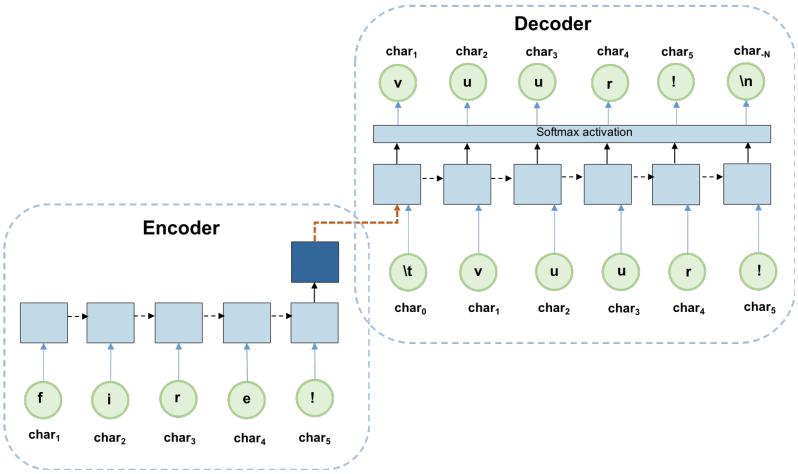


Figure 19: Seq2seq model without independence assumption on the output sequence

$$P(y_0, ...y_n|x_0, ...x_k) = P(y_0|x_0, ...x_k)...P(y_n|x_0, ...x_k) \tag{19}$$

In this case, computing the target, eq. (17), is reduced to greedily finding the y assignment that maximizes the distribution at that time, as shown in eq. (20).[11]

$$\underset{(y_0,...y_n)}{\arg\max}\, P(y_0, y_1, ..y_n|x_0, x_1, ...x_k) = \left(\underset{y_i}{\arg\max}\, P(y_i|x_0, x_1, ...x_k)\right)_{i=0,...,n} \tag{20}$$

In a more general setting, we cannot assume that the output sequence consist of independent symbols. We therefore condition the decoder on the previous selected output as shown in Figures 19 and 20b.

In this case, however, a greedy solution will not guarantee finding the sequence that maximizes the joint probability since selecting the most likely symbols early on can lead us to a path of unlikely symbols further in the sequence. To guarantee the maximum joint probability we would need to search through the whole space of output sequences. For long sequences this search would be computationally expensive.

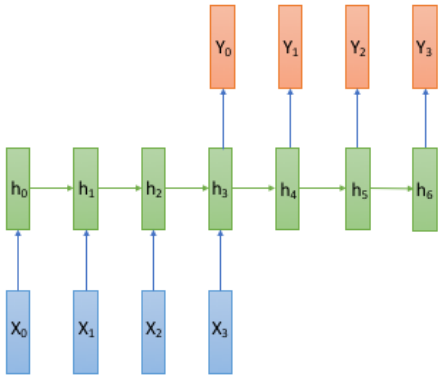
Sequence to sequence models — Beam Search

For each step in generating the output sequence we can branch-out in a number of branches equal to the number of possible output symbols. As such the search can well be structured as a tree, and tree search algorithms such as depth first or bread first search are applicable for solving eq. (17).

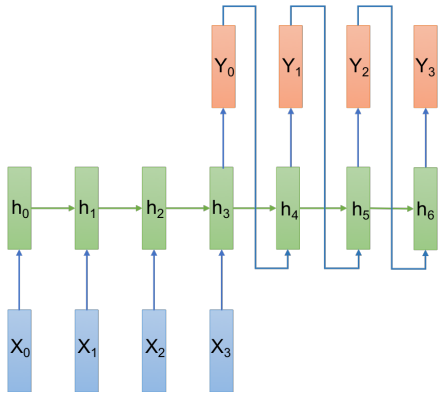
To avoid the high computational complexity we may relax the requirement for a maximum guarantee and use a heuristic approach that comes with a significantly lower computational cost as the sequence length increases. One commonly used search algorithm is Beam Search.[12]

[11] We assume independence of previous values, but not of index, so this optimal y needn't be the same at every index.

[12] Furcy, David, and Sven Koenig. "Limited discrepancy beam search." IJCAI. 2005



(a) Seq2seq model assuming independent outputs.



(b) Seq2seq model without independence assumption on the output sequence.

Figure 20: Seq2seq with and without the independence assumption.

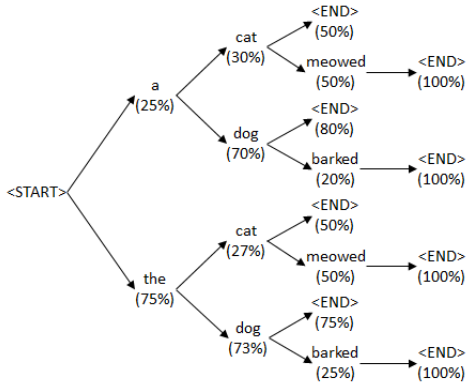


Figure 21: Generating a sequence - Beam Search

Beam search maintains a list of candidate sequences (a beam) that maximize the given metric and expands the search from that list rather that doing an exhaustive search — see Figure 21 for an illustration of this. In most practical applications the distribution of the solutions is such that Beam Search finds the most likely solution. That is why the algorithm is commonly used in combination with seq2seq models.

Sequence to sequence models — Example

To get a better intuition of the seq2seq models, let us consider a practical setting. The task is translation of a sentence from one language to another. We are given training data $D : (x, y)$ where x and y are sequences of words with different lengths in two different languages. In order to learn efficiently, we want to use meaningful representations of the words, so we use pre-trained word-embeddings. Alternatively we can train word embedding matrices in the model directly. Overall this gives us the architecture shown in Figure 22.

There are a number of decisions highlighted in this solution. The *encoding* produced by the encoder

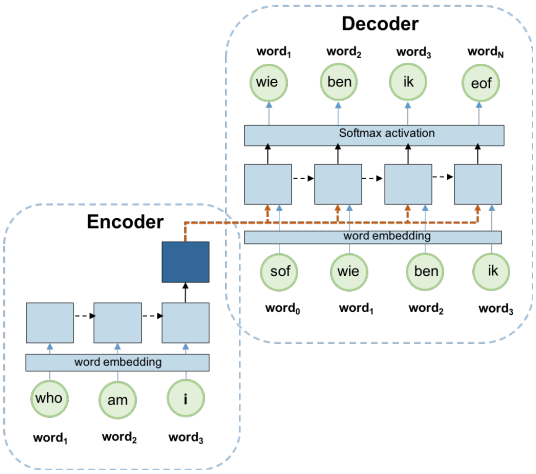


Figure 22: Seq2seq model for translation

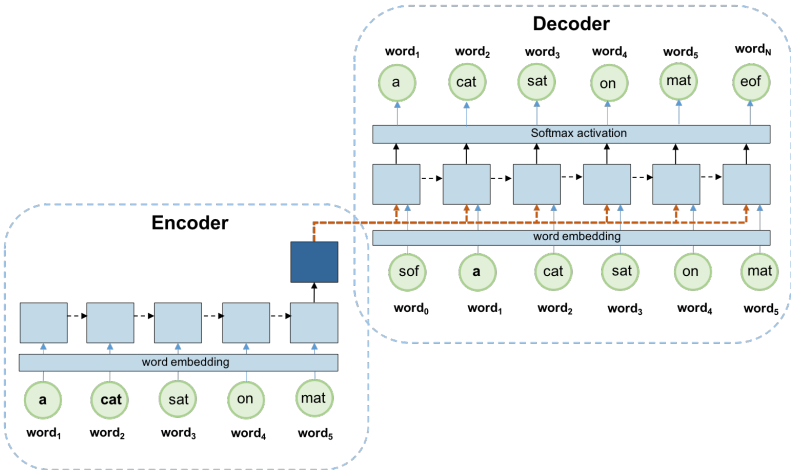


Figure 23: Seq2seq autoencoding

is presented to the decoder at each time step rather than only as initialization of its hidden state. The decoder is presented with the previous symbol (element) of the output at each decoding step. An embedding layer is present both for the encoder and decoder and is used to represent the words (symbols). Given that the words come from different vocabularies the embedding matrices are different.

Note: During inference the model provides a probability distribution over the possible sequences and we choose to use Beam Search to find a list of likely solutions. However, during training we typically only present the model with the solution given in the training (assuming we are given one target sequence per input sequence). Therefore, during training we do not run Beam Search for the decoder. Instead we compute the negative log-likelihood of the target sequence according to the computed distribution, and try to minimize this. [13]

An autoencoder for sequences using the same architectural choices is shown in Figure 23

[13]Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. "Sequence to sequence learning with neural networks." Advances in neural information processing systems. 2014.

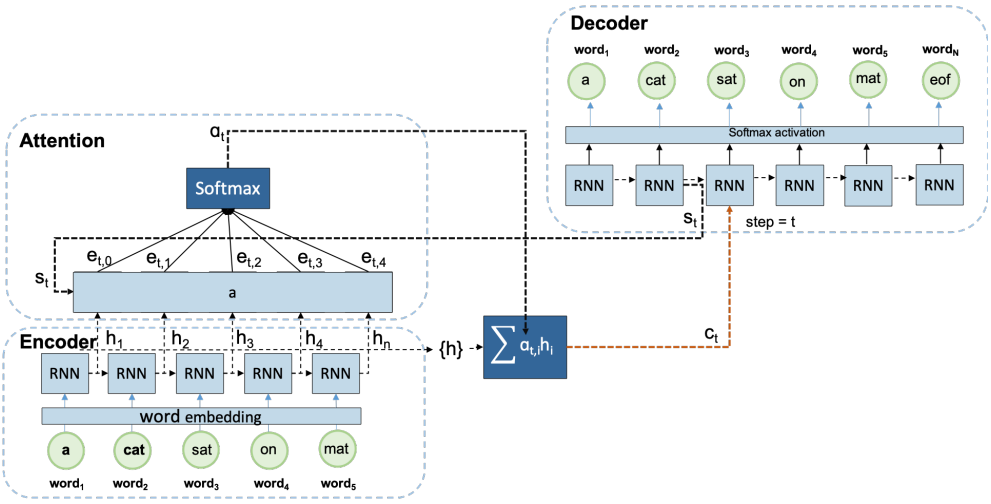


Figure 24: Seq2seq with attention

8.4 Attention Mechanism

Even though the seq2seq model is an efficient architecture for developing a sequence representation and mapping a sequence to another sequence, work in the field has brought up arguments that encoding the whole sequence to a single vector, as shown in Figure 22, may be challenging. Or in other words, finding the parameters for a seq2seq model with SGD and backprop may be difficult given the structure of their architecture.

Instead of using a single *code*, we can use a model with an "attention mechanism"[14]. The high-level idea of these models is that rather than using a code to store all the information about the input sequence, we should use it as a context and enable the decoder to have a look-back mechanism that gives it access to the input data.

Let use consider the following illustrative example. For the task of text summarizing, we are given one longer text sequence for which we need to produce a shorter text sequence. The shorter version of the text should capture the meaning of the longer one with fewer words. Suppose we develop a model for this task with the same architecture as in Figure 22. With this architecture the decoder model needs to be able to generate summarizing sequences of text based on the given *encoding* of the long sequence. However, when writing a summary, one can often copy certain words or short phrases from the original text. If the decoder had access to the original text, it could develop a mechanism for copying in addition to generation, which could make it more efficient or more precise.

In general the idea is that if the decoding model can 'attend' to the input sequence, this model can become more efficient, or we can become have more efficient training of such models in terms of the amount of training data or complexity of the model. This feature is referred to as the attention mechanism.

In Figure 24 we can see a seq2seq model with attention. In this model the decoder receives an

[14]Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." (2014).

additional input c_t at each time step. The vector c_t is a convex combination of the representation (or in this case the hidden state) of the input sequence produced by the encoder at each time step. So, now the decoder not only has access to the representation of the whole sequence but also to the individual elements of the input. The model also learns how to 'attend' to the individual inputs with different weights. In Figure 24, this is done using the α (attention) vector. The attention vector α is indexed by the time step t, meaning that the decoder 'attends' differently to the input at each time step.

The α is normalized to sum up to one, so we can view it as a probability distribution over the input. It is formed by a dense layer that inputs the encoding vectors h and a decoder state vector s (typically based on the hidden state of the decoder). The hidden layer produces the logit values of the distribution of the attention that is then normalized with a softmax activation function.

Let us look at how a model using this kind of attention works in detail.[15] We have an input sequence $\{x_0, x_1, ..., x_k\}$, and want to model $P(y_1, \ldots, y_n \mid x_1, \ldots, x_k)$. To do this, we first process the input sequence using a word embedding and a GRU[16] to come to hidden states $h_1, \ldots, h_k$, so

$$h_t = f(x_t, h_{t-1})$$

where f is the combination of a GRU cell and an embedding function.

The decoder too uses a GRU with hidden state s_k where s_0 is initialized as

$$s_0 = \tanh(W_s h_1).$$

The difference is that at every time step i, the decoding GRU is presented with a (different) context vector c_k. This context vector is computed as follows. First we compute the logits for the weights of the attention vector

$$e_{ij} = v_a^\top \tanh(W_a s_{i-1} + U_a h_j)$$

from which we compute the attention vector as

$$\alpha_k = \text{softmax}(e_k).$$

This attention vector is then used to compute the context vector as

$$c_i = \sum_j \alpha_{ij} h_j.$$

We can then feed the context vector, together with the hidden state and the previous output to the decoding GRU to get the next hidden state, and thu the next context vector

$$s_i = g(y_{i-1}, s_{i-1}, c_i).$$

To get the output, we can now feed the previous output, the hidden state, and the context vector to an MLP with softmax output:[17]

$$P(y_i \mid x_1, \ldots, x_k, y_1, \ldots, y_{i-1}) = \text{softmax}(W_o s_i + C_o c_i + U_o E y_{i-1})$$

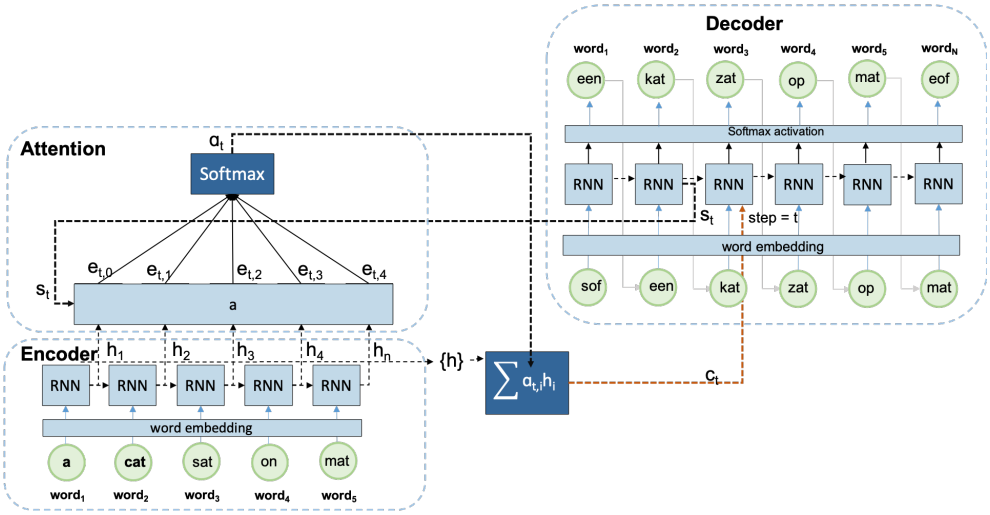


Figure 25: Attention mechanism

where E is an embedding matrix. This architecture is shown in Figure 25 for the different task of translating sentences from English to Dutch.

With this architecture, the decoder has a mechanism that allows it to access the (processed) input as an external memory. It can use the s_t signal as an addressing mechanism. In this way the model does not need to store all information needed for the decoding in the *encoding* of the input sequence, which can be particularly effective if the decoder needs to copy information from the input sequence.

Note: The weights of the attention vector α can also be informative of the operations of the model. Specifically, it can be interesting to look at what parts of the input was attended to for what parts of the output. This may be particularly useful in translation (or generally transducing) tasks, where a decision regarding an output word can be 'explained' by to what the model is attending to. Another perspective of the attention is as a means of alignment. The seq2seq models generally map sequences without indicating alignment. In other words, the model does not indicate how each output signal is aligned to an input symbol. For example in a event detection task, the input sequence may be a high dimensional signal that needs to be mapped to a sequence of events (e.g. ECG signal that is mapped to a sequence of Q, R, S-wave detections). In such a setting the attention can indicate alignment of the input to the output symbols that may be useful for diagnosing the model or even as a feature of the model.

[15]For simplicity we have omited writing biases.

[16]In the original paper on attention, a bi-directional GRU was used, but we want to keep the example simple.

[17]In the original paper some more processing was done to come to an output distribution, but this changes nothing to the working of the attention mechanism, so this is besides the point.