Neural Text Encoders

F

From Random Experiments to Natural Language Processing – by Wilker Aziz. This book has not been published yet.

ILOs

After completing this chapter, you should be able to

- specify neural network blocks ideal for manipulation of symbolic data

F.1 From Tokens to Vectors

NNs are functions that take real-valued vectors as inputs and produce real-valued vectors as outputs. They are quite flexible, but not flexible enough to process *symbolic* data without special treatment.

In this section we discuss techniques that we can use to map from a symbolic space, such as a finite countable set (*e.g.*, the vocabulary of a language) to a real coordinate space (a real-valued vector space of fixed dimensionality).

Vocabulary

The first thing we do when working with text is to map our tokens to unique (often 0-based) integer identifiers (ids). It does not matter which symbol gets which id, so long as the correspondence between tokens and ids is fixed and unique (see the example in Table F.1).

From now on, if we talk about a *token* we mean a *token id* (*i.e.*, the 0-based integer that identifies that symbol uniquely).

Table F.1: A vocabulary of known symbols and their unique 0-based integer identifiers. The first two symbols are reserved for special use, the remaining ones are words in our toy language.

id	symbol
0	`[PAD]`
1	`[UNK]`
2	`are`
3	`awesome`
4	`cats`
5	`cute`
6	`dogs`

One-Hot Encoding

If we know a *finite* set $\mathcal{V}$ of tokens (*e.g.*, words), and the total number of unique symbols in it is some number $V = |\mathcal{V}|$, then the simplest technique to map tokens to vectors is to map each token $t \in \mathcal{V}$ to a vector $\mathbf{v} = \text{onehot}_V(t)$ such that $\mathbf{v} \in \mathbf{R}^V$, $v_t = 1$ and $v_{d \neq t} = 0$.

This technique is called *one-hot encoding* because it returns a vector whose coordinates are 0 for all but one dimension (that which indicates the token we are encoding), which gets 1.

A vocabulary often contains some special symbols, which help us design good text encoders. For example, `[UNK]` is used in place of unknown words: if "otters" was never seen in training, but occurs at one point in `otters are cute`, we change that document to `[UNK] are cute`. When working with batches of documents of different length, we extend shorter documents to match the length of the longest one, so they can be stacked together into something like a table:

```
cats  and   dogs  are   awesome
cute  cats  [PAD] [PAD] [PAD]
[UNK] are   cute  [PAD] [PAD]
```

where the special symbol `[PAD]` identifies cells that are not part of any document.

Using the vocabulary from Table F.1, $\text{onehot}_7(\texttt{awesome}) = (0, 0, 0, 1, 0, 0, 0)^\top$.

Word Embedding

Our next operation is a bit more interesting. It allows us to associate with each token a vector of *trainable parameters* whose dimensionality we choose independently of V. So, instead of encoding a token $t \in \mathcal{V}$ into a V-dimensional one-hot vector, we encode it into a D-dimensional vector

of real-numbers (and we choose D). We normally refer to this operation as *embedding* the token into a D-dimensional space.

Suppose we have a table of parameters $\mathbf{E} \in \mathbb{R}^{V \times D}$, with one D-dimensional row for each of the known symbols in the vocabulary $\mathcal{V}$. Then, for some symbol $t \in \mathcal{V}$, the embedding operation

$$\mathbf{e} = \text{embed}_D(t; \mathbf{E}) \tag{F.1}$$

returns the vector $\mathbf{e} \in \mathbb{R}^D$ that corresponds to it. The subscript $_D$, in the operation, indicates the output dimensionality. After ';' we have the trainable parameters of the operation.

If we had a sequence of symbols, for example, a document $w_{1:l}$ where $w_i \in \mathcal{V}$, we could apply the embedding operation to each symbol in the sequence, and denote it this way:

$$\mathbf{e}_i = \text{embed}_D(w_i; \mathbf{E}) \quad \text{for } i = 1, \ldots, l\ . \tag{F.2}$$

Then, the sequence $\mathbf{e}_{1:l}$ contains l vectors, each D-dimensional, one for each of the tokens in $w_{1:l}$.

Table F.2: A table of 3-dimensional embeddings for the tokens in Table F.1. Example generated via `rn.normal(size=(7, 3))` with `rng = np.random.RandomState(42)`, we round the results to 2 decimals for visualisation purposes.

0.5	−0.14	0.65
1.52	−0.23	−0.23
1.58	0.77	−0.47
0.54	−0.46	−0.47
0.24	−1.91	−1.72
−0.56	−1.01	0.31
−0.91	−1.41	1.47

Using the vocabulary of Table F.1 and $\mathbf{E}$ as defined in Table F.2, $\text{embed}_3(\texttt{dogs}; \mathbf{E})$ returns $(-0.91, -1.41, 1.47)^\top$.

Most NN operations, employ trainable parameters, which help us realise the deterministic mapping from input to output. Our notation hosts these parameters after the semicolon `;`. When describing operations, we may choose to 'name' the set of parameters, for example, in order to distinguish two instances of the same operation, each having its own parameter set. Imagine we are working with two languages (*e.g.*, translating from one to another), then we may need to work with two vocabularies and embed tokens from each vocabulary independently. We could denote the operation for one language by $\text{embed}_D(\cdot; \theta_{\text{L1}})$ and use $\text{embed}_D(\cdot; \theta_{\text{L2}})$ for the other.

F.2 Pooling from Multiple Vectors

Sometimes we need to combine a variable number of D-dimensional vectors into a single D-dimensional vector, this is usually referred to as *pooling*. There are different pooling operations, some with and some without trainable parameters.

- Input: a collection $\mathbf{e}_1, \ldots, \mathbf{e}_l$ of $l > 0$ vectors, all D-dimensional.
- Output: a single D-dimensional vector $\mathbf{u} \in \mathbf{R}^D$.

Batched implementations of pooling operations must give special treatment to positions that should not affect the result (*i.e.*, those that correspond to `[PAD]`).

Sum Pooling

The output $\mathbf{u} = \sum_{i=1}^{l} \mathbf{e}_i$ is the elementwise sum of the inputs. That is, for each dimension $d \in [D]$ in the output, we have

$$u_d = \sum_{i=1}^{l} e_{i,d}\ . \tag{F.3}$$

For a *batched* implementation, we can associate padded input positions with a $\mathbf{0} \in \mathbf{R}^D$ vector, as zeroes do not affect summation.

Average Pooling

The output $\mathbf{u} = \frac{1}{l}\sum_{i=1}^{l}\mathbf{e}_i$ is the elementwise average of the inputs. That is, for each dimension $d \in [D]$ in the output, we have

$$u_d = \frac{1}{l}\sum_{i=1}^{l} e_{i,d} \ . \tag{F.4}$$

For a *batched* implementation, we can associate padded input positions with a $\mathbf{0} \in \mathbf{R}^D$ vector, as zeroes do not affect summation, and we have to count length correctly (*i.e.*, discounting pad tokens) for each input sequence in the batch.

Max Pooling

The output $\mathbf{u}$ is the elementwise maximum of the inputs. That is, for each dimension $d \in [D]$ in the output, we have

$$u_d = \max_{i\in[l]} e_{i,d} \ . \tag{F.5}$$

For a *batched* implementation, we can associate padded input positions with a $-\infty \in \mathbf{R}^D$ vector, as negative infinite does not affect maximisation.

F.3 Mapping from One Real Coordinate Space to Another

Sometimes we are working with vectors of a certain dimensionality I and we need to convert them to vectors of another dimensionality O. This is very common, for example, when mapping a document encoding (*e.g.*, average embedding, or bag-of-words encoding) to C real-valued scores (or logits).

Linear Transformation

A *linear transformation* is the standard way to get this done: with trainable parameters $\mathbf{W} \in \mathbb{R}^{O\times I}$ and $\mathbf{b} \in \mathbb{R}^O$ we can map any I-dimensional input $\mathbf{u}$ to an output $\mathbf{v} \in \mathbf{R}^O$ via $\mathbf{v} = \mathbf{W}\mathbf{u} + \mathbf{b}$.

The linear transformation can be compactly denoted by:

$$\mathbf{v} = \text{linear}_O(\mathbf{u}; \mathbf{W}, \mathbf{b}) \ . \tag{F.6}$$

In some situations, we prefer to use this transformation without a bias vector (*i.e.*, setting $\mathbf{b}$ to a vector of 0s); sometimes, this is also called a *projection*.

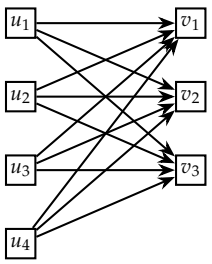

Figure F.1: Linear transformation mapping $I = 4$ input units to $O = 3$ output units. The parameters of the network are: $\theta\{\mathbf{W} \in \mathbb{R}^{O\times I}, \mathbf{b} \in \mathbb{R}^O\}$, which are used to map from I input units to O output units via $\mathbf{W}\mathbf{u} + \mathbf{b}$.

Nonlinear Activation Functions

Sometimes we need to work on a *subspace* of the real coordinate space, for example, where numbers are constrained to being positive, or strictly positive, or strictly positive and sum up to 1, etc. We can achieve this by working with *activation functions*. An activation function will not change the dimensionality of its input, and it does not require any trainable

parameter, typically, an activation function is a formula that transforms a vector elementwise.

For example, if $\mathbf{u} \in \mathbb{R}^D$

- $\exp(\mathbf{u})$ applies $\exp(u_d)$ to each coordinate u_d of $\mathbf{u}$, hence returning a vector of D strictly positive numbers;
- $\text{relu}(\mathbf{u})$ applies $\max(0, u_d)$ to each coordinate u_d of $\mathbf{u}$, hence returning a vector of D positive (and possibly 0) numbers;
- $\tanh(\mathbf{u})$ applies $\tanh(u_d)$ to each coordinate u_d of $\mathbf{u}$, hence returning a vector D numbers in the space $(-1, 1)$;
- $\text{sigmoid}(\mathbf{u})$ applies $\frac{1}{1+\exp(u_d)}$ to each coordinate u_d of $\mathbf{u}$, hence returning a vector of D independently normalised probability values (*i.e.*, each in the space $(0, 1)$);
- $\text{softplus}(\mathbf{u})$ applies $\log(1 + \exp(u_d))$ to each coordinate u_d of $\mathbf{u}$, hence returning a vector of D strictly positive numbers;
- $\text{softmax}(\mathbf{u})$ applies $\frac{\exp(u_d)}{\sum_{k=1}^{D} \exp(u_k)}$ to each coordinate u_d of $\mathbf{u}$, hence returning a vector of D strictly positive numbers that add up to 1 (*i.e.*, a point in the simplex $\Delta_{D-1} \subset \mathbb{R}^D$).

The *sigmoid* function is also known as the *logistic* function (in statistics). In various reference texts, the *sigmoid* function is denoted by $\sigma(\cdot)$, that is, $\sigma(a) = \frac{1}{1+\exp(a)}$. Here are some useful results about the sigmoid function:

- $\sigma(a) = \frac{1}{1+\exp(a)}$
  $= \frac{\exp(u_d)}{1+\exp(u_d)}$
  $= 1 - \sigma(-a)$
- $\frac{\mathrm{d}}{\mathrm{d}a}\sigma(a) = \sigma(a) \times (1 - \sigma(a))$

The *softplus* function can be thought of as a smooth approximation to the relu function. It is often used when we need strictly positive numbers while also needing more numerical stability than the exp function can offer. In particular, the derivative of the exp function is the exp function (*i.e.*, $\frac{\mathrm{d}}{\mathrm{d}a}\exp(a) = \exp(a)$), while the derivative of the softplus function is the sigmoid function: $\frac{\mathrm{d}}{\mathrm{d}a}\text{softplus}(a) = \text{sigmoid}(a)$, the advantage being that exp can become very large, while sigmoid is always between 0 and 1.

F.4 Composing Multiple Vectors

We now look into *composition functions* that combine multiple vectors of fixed dimensionality into one (or more) vectors, while possibly changing the dimensionality of the output with respect to the dimensionality of the input(s). While pooling functions are necessarily discarding some information available in the input (*e.g.*, sum discards order in the input collection, average discards order and size of the input collection, maximum discards inputs that are not the largest, etc.), we compose vectors to a) *not* discard anything important, and b) let the inputs *interact* to create new, more complex features.

Concatenation

The simplest thing we can do, in order not to discard *any* information, is to concatenate input vectors in their given order. For example, we can concatenate an I_1-dimensional vector $\mathbf{u}$ with an I_2-dimensional vector $\mathbf{v}$ obtaining an $I_1 + I_2$-dimensional vector:

$$\mathbf{h} = \text{concat}(\mathbf{u}, \mathbf{v}) = (u_1, \ldots, u_{I_1}, v_1, \ldots, v_{I_2})^\top . \tag{F.7}$$

A few things to consider about concatenation. It does not require any trainable parameters, and can be applied to any number of input vectors. However, the more inputs we have, the more outputs we have. If in a certain context we need to deal with variable-length inputs, we would then have to deal with variable-length outputs, which is sometimes not possible. Besides, and perhaps most importantly, while not discarding any information, concatenation is unable to create new features (*e.g.*, by letting input features interact), this, however, we address next.

Another way to denote the concatenation of l vectors $\mathbf{u}_1, \ldots, \mathbf{u}_l$, each of dimensionality $\dim(\mathbf{u}_i)$, is to write $[\mathbf{u}_1, \ldots, \mathbf{u}_l]$, the output vector will then have dimensionality $\sum_{i=1}^{l} \dim(\mathbf{u}_i)$.

Feed-Forward Network

The simplest feed-forward network (FFNN) combines two linear transformations with a non-linear activation function in between. The input features 'interact' forming the intermediate ('hidden') features. It is also possible to make an FFNN *deeper*, by stacking additional linear transformations (again, with nonlinearities in between).

This is the simplest example:

$$\mathbf{h} = a(\text{linear}_H(\mathbf{u}; \theta_{\text{hid}})) \tag{F.8a}$$

$$\mathbf{v} = \text{linear}_O(\mathbf{h}; \theta_{\text{out}}) \tag{F.8b}$$

where $a(\cdot)$ is an elementwise nonlinearity (*e.g.*, those in Subsection F.3), typically tanh or relu, θ_{hid} are the parameters of the first input-to-hidden layer (a weight matrix and a bias vector) and θ_{out} are the parameters of the second hidden-to-output layer (another weight matrix and another bias vector). See that instead of writing down the parameters explicitly, we named them (using θ and a suggestive subscript); this is a convenient notation shortcut.

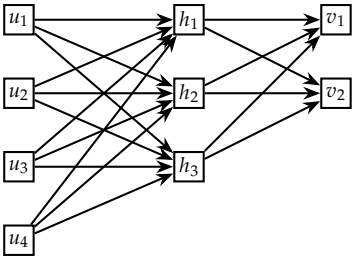

Figure F.2: Feed-forward network mapping $I = 4$ input units to $H = 3$ hidden units and then $O = 2$ output units. The parameters of the network are: $\theta_{\text{hid}} = \{\mathbf{W}^{[1]} \in \mathbb{R}^{H \times I}, \mathbf{b}^{[1]} \in \mathbb{R}^{H}\}$, which are used to map from I input units to H hidden units via $a(\text{linear}_H(\mathbf{u}; \theta_{\text{hid}}))$, and $\theta_{\text{out}} = \{\mathbf{W}^{[2]} \in \mathbb{R}^{O \times H}, \mathbf{b}^{[2]} \in \mathbb{R}^{O}\}$, which are used to map from H hidden units to O output units via $\text{linear}_O(\mathbf{u}; \theta_{\text{out}})$.

Whenever we want to compose a *fixed* number of input vectors into a single output vector, it is very common to combine a concatenation operation and an FFNN block. For example, if we want to compose K vectors $\mathbf{u}_1, \ldots, \mathbf{u}_K$, each of dimensionality $\dim(\mathbf{u}_k)$, we: concatenate inputs to form a vector $\mathbf{c}$ of dimensionality $\sum_{k=1}^{K} \dim(\mathbf{u}_k)$

$$\mathbf{c} = [\mathbf{u}_1, \ldots, \mathbf{u}_K] , \tag{F.9a}$$

transform it into H nonlinear hidden units

$$\mathbf{h} = a(\text{linear}_H(\mathbf{c}; \theta_{\text{hid}})) , \tag{F.9b}$$

and then transform the result into O output units:

$$\mathbf{v} = \text{linear}_O(\mathbf{h}; \theta_{\text{out}})) . \tag{F.9c}$$

Recurrent Neural Network

A recurrent neural network (RNN) uses one or more feed-forward networks inside of a for-loop to iterate over the steps of a sequence, at each step the RNN combines a *recurrent* state and the input at that step using a FFNN. Rather than one different FFNN per step, the RNN reuses the same FFNN. Suppose we have a sequence of vectors $\mathbf{e}_{1:l}$, all of which are I-dimensional, for example, those are the the token embeddings for a document $w_{1:l}$.

The RNN has a initial state $\mathbf{u}_0$, this is an H-dimensional vector of trainable parameters. Then, at each step $i \in [l]$ of the sequence, the RNN composes the preceding recurrent state $\mathbf{u}_{i-1}$ with the current input $\mathbf{e}_i$, this combination produces a new value $\mathbf{u}_i$ for the recurrent state, which will be used in the next step. See the Figure F.3 for an illustration.

We can describe the computation as follows:

$$\mathbf{u}_i = \text{rnnstep}_H(\mathbf{u}_{i-1}, \mathbf{e}_i; \theta) \quad \text{for } i \in [l]. \tag{F.10}$$

The simplest implementation of the block 'rnnstep' looks like this:

$$\text{rnnstep}_H(\mathbf{u}_{i-1}, \mathbf{e}_i; \theta) = \tanh(\mathbf{R}\mathbf{u}_{i-1} + \mathbf{W}\mathbf{e}_i) \tag{F.11}$$

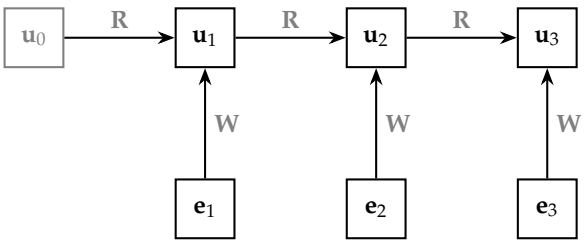

Figure F.3: Recurrent neural network unfolded for 3 steps. In gray, we depict the parameters of the RNN: the initial state $\mathbf{u}_0 \in \mathbb{R}^H$ and the matrices $\mathbf{R} \in \mathbb{R}^{H \times H}$ and $\mathbf{W} \in \mathbb{R}^{H \times I}$. At each step $i \in [l]$ of the process, the RNN uses the same two matrices ($\mathbf{W}$ and $\mathbf{R}$) to compose a recurrent state $\mathbf{u}_i$ as a function of the preceding recurrent state $\mathbf{u}_{i-1}$ and the ith input $\mathbf{e}_i$. One of the simplest RNN cells uses the transformation: $\mathbf{u}_i = \tanh(\mathbf{R}\mathbf{u}_{i-1} + \mathbf{W}\mathbf{e}_i)$.

where the trainable parameters $\theta = \{\mathbf{u}_0 \in \mathbb{R}^H, \mathbf{R} \in \mathbb{R}^{H \times H}, \mathbf{W} \in \mathbb{R}^{H \times I}, \}$ are two matrices that project the recurrent state and the current input to size H. This is even simpler than employing a full FFNN, since we only have a hidden layer.

If you pay close attention to the illustration, or to the formulae, you will see that the recurrent state at any one position i can potentially store information from any of the inputs $\mathbf{e}_1, \ldots, \mathbf{e}_{i-1}$. Besides, due to the nonlinearity of the rnnstep function, the state $\mathbf{u}_i$ is sensitive to the order in which the inputs are presented (that is, if we presented inputs in a different order, the numerical values of the coordinates of $\mathbf{u}_i$ might differ). This is an important aspect of a feature function for natural language processing, given that natural languages encode a lot of information in word order.

This form of RNN is also called an RNN *encoder*, in allusion to the fact that $\mathbf{u}_i$ encodes the vector sequence $\mathbf{e}_{1:i}$. In written form, a call to an RNN encoder can be denoted even more compactly as

$$\mathbf{u}_{1:l} = \text{rnnenc}_H(\mathbf{e}_{1:l}; \theta) \,. \tag{F.12}$$

This function takes a sequence of vectors as input and returns a sequence of H-dimensional vectors, each encoding a longer subsequence of the input sequence. The last vector $\mathbf{u}_l$ of the output sequence has the potential to store information about the entire input sequence (in its given order).

Long Short-Term Memory

The simplest RNN suffers from certain instability issues (they struggle to retain information from long sequences and they lead to numerical problems in optimisation). A modern RNN-type architecture that does not exhibit these problems is the Long Short-Term Memory. It is not necessary, for the applications in this book, to study the LSTM paper. For completeness, we briefly explain the internal design of the LSTM here, but this level of detail is more than what we need in this book. The choice of letters we use in this part are internal to the LSTM and are not to be confused for letters used in other contexts.

For a step t, let $\mathbf{e}_t$ be an I-dimensional input to an LSTM (*e.g.*, this may be an embedding for the token w_t in a document).

At this point, the memory of an LSTM is made of two K-dimensional vectors called the *cell vector* $\mathbf{c}_{t-1}$ and the *hidden state* $\mathbf{h}_{t-1}$, each of which is K-dimensional. When we process the input $\mathbf{x}_t$ with an LSTM, these two vectors are updated step by step as shown below:

$$\mathbf{i}_t = \text{sigmoid}(\text{linear}_K(\mathbf{e}_t; \theta_1) + \text{linear}_K(\mathbf{h}_{t-1}; \theta_2)) \tag{F.13a}$$
$$\mathbf{f}_t = \text{sigmoid}(\text{linear}_K(\mathbf{e}_t; \theta_3) + \text{linear}_K(\mathbf{h}_{t-1}; \theta_4)) \tag{F.13b}$$
$$\mathbf{g}_t = \tanh(\text{linear}_K(\mathbf{e}_t; \theta_5) + \text{linear}_K(\mathbf{h}_{t-1}; \theta_6)) \tag{F.13c}$$
$$\mathbf{o}_t = \text{sigmoid}(\text{linear}_K(\mathbf{e}_t; \theta_7) + \text{linear}_K(\mathbf{h}_{t-1}; \theta_8)) \tag{F.13d}$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t \tag{F.13e}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{F.13f}$$

The first four steps compute the following using the input and the hidden state: the *input gate* $\mathbf{i}_t$, then the *forget gate* $\mathbf{f}_t$, the *draft cell* $\mathbf{g}_t$, and the *output gate* $\mathbf{o}_t$. These are all K-dimensional, and the linear transformations all have their own parameters (there 8 such affine transformations in total, they map either from I dimensions to K dimensions, or from K dimensions to K dimensions, and they have biases vectors in them). The last couple of steps finally update the cell and the hidden state by combining the intermediate gates and draft cell. The symbol $\odot$ denotes elementwise multiplication. After the update the LSTM memory is made of two states $\mathbf{c}_t$ and $\mathbf{h}_t$, each K-dimensional.

Typically, we regard the cell states $\mathbf{c}_{1:l}$ as something internal to the LSTM and we rarely need to use them for anything outside of it. It is the hidden state $\mathbf{h}_i$ at each step that we normally want to use in applications (*e.g.*, as a representation of the sequence $\mathbf{e}_{1:i}$). Hence, in this book, it is sufficient to think of the LSTM encoder as a function

$$\mathbf{h}_{1:l} = \mathrm{lstm}_K(\mathbf{e}_{1:l}; \theta) \tag{F.14}$$

that returns a sequence $\mathbf{h}_{1:l}$ of hidden states encoding the input sequence $\mathbf{e}_{1:l}$. This also shows that an LSTM is indeed just a special type of RNN.

Bidirectional RNN

When we encode a document, for example, in text classification, we have the entire document available to us and we are by no means constrained to processing the words from left-to-right. Why not encode it from right-to-left, for example?

Even better, why not do both? This way, whenever we look at a given position i, we can obtain information from its left (*i.e.*, from $\mathbf{e}_1, \ldots, \mathbf{e}_i$) and from its right (*i.e.*, from $\mathbf{e}_i, \ldots, \mathbf{e}_l$). This gives us a fully contextualised view of the token that sits at the ith position of a document.

An RNN cell, by design, makes computations in a single direction (*e.g.*, left-to-right), but we can use 2 different RNN cells, one that reads the sequence in one order and another that reads the sequence in reversed order.

We do not need to invent a new RNN cell for this, we can simply reverse the inputs to a standard RNN cell:

$$\mathbf{r}_{1:l} = \mathrm{reverse}(\mathbf{e}_{1:l}) \tag{F.15a}$$

$$\mathbf{v}_i = \mathrm{rnnstep}_K(\mathbf{v}_{i-1}, \mathbf{r}_i; \theta_{\mathrm{renc}}) \,. \tag{F.15b}$$

Because the inputs have been reversed in order. For example, in a sentence of length $l = 10$, $\mathbf{v}_2$ knows about w_9 through $\mathbf{r}_2$ and about $w_{>9}$ through $\mathbf{v}_1$. The last state $\mathbf{v}_l$ has information about the entire document $w_{1:l}$, but processed it in reversed order.

Therefore a reversed RNN encoder can be denoted compactly as follows:

$$\mathbf{v}_{1:l} = \mathrm{rnnenc}_K(\mathrm{reverse}(\mathbf{e}_{1:l}); \theta_{\mathrm{renc}}) \tag{F.16}$$

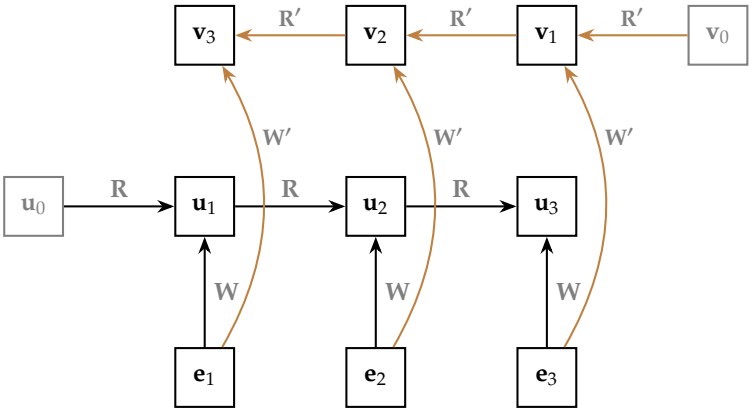

Figure F.4: Bidirectional recurrent neural network (BiRNN) unfolded for 3 steps. The parameters of the architecture are shown in gray: $\mathbf{u}_0 \in \mathbb{R}^H$ and the matrices $\mathbf{R} \in \mathbb{R}^{H \times H}$ and $\mathbf{W} \in \mathbb{R}^{H \times I}$ are the parameters of the forward RNN (which processes the sequence from left-to-right, as shown by the black arrows); $\mathbf{v}_0 \in \mathbb{R}'^H$ and the matrices $\mathbf{R}' \in \mathbb{R}^{H \times H}$ and $\mathbf{W}' \in \mathbb{R}^{H \times I}$ are the parameters of the backward (or reverse) RNN (which processes the sequence from right-to-left, as shown by the brown arrows). The output is a sequence of concatenated states: $\langle [\mathbf{u}_0, \mathbf{v}_0], [\mathbf{u}_1, \mathbf{v}_3], [\mathbf{u}_2, \mathbf{v}_2], [\mathbf{u}_3, \mathbf{v}_1] \rangle$.

Note that we named the parameter set differently: θ_{enc} for the first RNN cell, and θ_{renc} for the second one, that is because we indeed want to have two different sets of parameters. If we used the same set of parameters for both directions, that probably would not work very well, as reading in one direction and reading in another are conceptually two different operations.

The *bidirectional RNN encoder* (or BiRNN) is our preferred text encoder (in particular, when the RNN cells are LSTMs, and we get a BiLSTM), it can be denoted as follows:

$$(\mathbf{o}_{1:l}, \mathbf{z}) = \text{birnn}_{2K}(\mathbf{e}_{1:l}; \theta_{\text{enc}} \cup \theta_{\text{renc}}) \tag{F.17}$$

and here are the operations that it performs:

$$\mathbf{u}_{1:l} = \text{rnnenc}_K(\mathbf{e}_{1:l}; \theta_{\text{enc}}) \tag{F.18a}$$

$$\mathbf{v}_{1:l} = \text{rnnenc}_K(\text{reverse}(\mathbf{e}_{1:l}); \theta_{\text{renc}}) \tag{F.18b}$$

$$\mathbf{o}_i = \text{concat}(\mathbf{u}_i, \mathbf{v}_{l-i+1}) \qquad \text{for } i \in \{1, \dots, l\} \tag{F.18c}$$

Its outputs are $2K$-dimensional because after processing the sequence from left-to-right with the first RNN encoder and from right-to-left with the second RNN encoder, it then concatenates the two views of the process in such a way that $\mathbf{o}_i$ has information about w_i, $w_{<i}$ and $w_{>i}$. The vector $\mathbf{z} \in \mathbb{R}^{2K}$ is the concatenation of the last LSTM state going right with the last LSTM state going left and, hence, stands as a single-vector representation of the complete sequence.

See Figure F.4 for an illustration of how the two RNN cells can be used to obtain the bidirectional RNN encoder: