RNN in action: Understanding sequences

We will study a toy example that is typical of many tasks involving sequences

- Given a prefix of a sequence
- Predict the next element

For example

- Predict the next word in a sentence
- Predict the next price in a timeseries of prices

Being able to predict the next element may be key to understanding the "logic" underlying a sequence

- You have to understand context and domain
- You have to understand how earlier elements influence latter elements

Predict the next: Data preparation

It is our belief that Machine Learning is a *process* and not just a collection of models.

We have recently been emphasizing the models but let's review the process.

Recipe for Machine Learning

It is usually the case that Sequence data involves substantial Data Preparation.

Suppose our task is to predict the next word in a sentence.

We are given (or must obtain) a collection of sentences (e.g., one or more documents) as our raw data.

But a sentence is not the format required for the training set of the "Predict the next word" task.

Data preparation is usually a substantial prerequisite for solving tasks involving sequences.

To be precises, the "Predict the next word" task involves

- Training a many to one RNN with examples created from a sequence.
- The elements of a single example are the prefix of a sentence
- The target of the example is the next word in the sentence

Let

$$[\ \mathbf{s}_{(t)} | 1 \leq t \leq T\ ]$$

be the sequence of words in sentence $\mathbf{s}$.

We will prepare $(T-1)$ examples from this single sentence.

$\langle \mathbf{X}, \mathbf{y} \rangle =$

i	$\mathbf{x^{(i)}}$	$\mathbf{y^{(i)}}$
1	$\mathbf{s}_{(1)}$	$\mathbf{s}_{(2)}$
2	$\mathbf{s}_{(1),(2)}$	$\mathbf{s}_{(3)}$
$\vdots$		
i	$\mathbf{s}_{(1),\ldots,(i)}$	$\mathbf{s}_{(i+1)}$
$\vdots$		
$(T-1)$	$\mathbf{s}_{(1),\ldots,(T-1)}$	$\mathbf{s}_{(T)}$

For example

$\mathbf{s} =$ "I am taking a class in Machine Learning"

i	$\mathbf{x^{(i)}}$	$\mathbf{y^{(i)}}$
1	[I]	am
2	[I, am]	taking
3	[I, am, taking]	a

Predict the next: data shape

We had warned earlier about the explosion of the number of dimensions of our data. Now is a good time to take stock

- $\mathbf{X}$, the training set, is a matrix with m rows
- Each row is an example $\mathbf{x^{(i)}}$
- Each example is a sequence $[\ \mathbf{x}^{\mathbf{(i)}}_{(t)} \mid 1 \le t \le ||\mathbf{x^{(i)}}||\ ]$
- Each element $\mathbf{x}^{\mathbf{(i)}}_{(t)}$ of the sequence encodes a word
- A word is encoded as a One Hot Encoded binary vector of length $||V||$ where V is the set of words in the vocabulary

Target $\mathbf{y^{(i)}}$ is also a word (so is vector of length $||V||$).

- Many to one: target is *not* a sequence

Predict the next: training

Just like training any other type of layer, but more expensive

- Each example involves multiple time steps: forward pass is time consuming
- The derivatives (needed for Gradient Descent) are more complex; backward pass complex and time consuming

Remember:

- the target $\mathbf{y}_{(t)}$ for step t should be $\mathbf{x}_{(t+1)}$ the next input
$$\mathbf{y}_{(t)} = \mathbf{x}_{(t+1)}$$

Predict the next

RNN as a generative model (fun with RNN’s)

The "Predict the next" word task is interesting on its own

- But a slight twist will make it extremely interesting

Suppose we have trained our model on a large collection of sentences of the same type (e.g., same author).

At test time, we feed a short "seed" sentence

$$\mathbf{x}_{(0)}, \ldots, \mathbf{x}_{(t)}$$

into the model and have it generate output.

But we then feed the output back into the model as input !

$$\mathbf{x}_{(t'+1)} = \mathbf{y}_{(t')} \text{ for } t' \geq t$$

- as in the Decoder in our Language Translation example

Test time: no forcing

The model would generate new text ad infinitum

- The next word generated would be based on what the model has learned from training
- To be the most probable word to follow the prefix

Voila: the RNN can *generate* text in the same style as the training sentences.

Using Machine Learning to *create* data is called *generative.*

Using Machine Learning to classify/predict (as we've been doing thus far) is called *discriminative.*

Generating strange things

Generating stories from seeds was very popular a few years back.

Let's look at some examples.

But first, a surprise:

- Rather than solving a "predict the next word" task
- All of the following examples were generated by a "predict the next **character**" task !

It is somewhat amazing that what is generated

- Has correctly spelled words/keywords
- Is Syntactically correct (sentences end with a ".", parentheses/brackets are balanced)
- Is meaningful: the elements/words are arranged in a logical order

Even though

- We have not explicilty identified any of these concepts
- Nor forced training to respect them (via a loss function)

Remember

- All of this behavior was "learned" by identifying the correct next **character**

- Fake Shakespeare (http://karpathy.github.io/2015/05/21/rnn-effectiveness/#shakespeare), or fake politician-speak
- Fake code
- Fake math textbooks (http://karpathy.github.io/2015/05/21/rnn-effectiveness/#algebraic-geometry-latex)
- Click bait headline generator (http://clickotron.com/about)

Training the generative model

Let's describe this generative process in more detail.

First: training a model.

At test-time (when we are generating new text) the outputs are fed back as next-step inputs

Test time: no forcing

But that's **not** how the model is trained

- the "next step" input is the **true** next element in the sequence
- this is how we construct the training data set

Training, with Teacher Forcing

This is called *teacher forcing*

$$\mathbf{x}^{(\mathbf{i})}_{(t)} = \mathbf{y}_{(t-1)}$$

rather than

$$\mathbf{x}^{(\mathbf{i})}_{(t)} = \hat{\mathbf{y}}_{(t-1)}$$

for $t > t'$.

- When extending the sequence
- A teacher forces the student (model) to continue with the *correct* answer
- Rather than the student's answer
- If it didn't do so, once the student (model) predicted incorrectly, it's errors would compound

Sampling from the generative model

Remember that a Classifier (the output stage of our model)

- generates a *probability distribution* (over the elements of the vocabulary V)

For the prediction, we usually *deterministically* choose the element of V with highest probability

$$\hat{\mathbf{y}} = \operatorname*{argmax}_{v \in V} p(v)$$

Deterministic choice might not be best for the generative process

- One wrong choice propagates to all successive elements of the sequence
- The output is always the same ! Boring !

So what is usually done is that our prediction is a *sample* from the probability distribution.

Summary

Here is the process in pictures

- The training inputs are given in red
- The test (inference) time inputs are given in black

Teacher forcing is indicated in red

- Predictions $[\ \hat{\mathbf{y}}_{(t)} \mid 1 \le t \le T\ ]$ **are not** used as input (lower right)
- Only correct targets $[\ \mathbf{y}_{(t)} \mid 1 \le t \le T\ ]$ are used

Sequence to Sequence: training (teacher forcing)

The input sequence to the Decoder is modified by

- prepending a special "start of output" symbol
$$\mathbf{x}_{(-1)} = \langle \text{START} \rangle$$
- appending a special "end of output" symbol $\langle \text{END} \rangle$ to training examples
    - The Decoder stops when it generates the end of output symbol

The *Encoder* is a many to one RNN

- Takes the variable length "seed" sequence
- Outputs a fixed length representation of the seed
    - This is one of the strengths of an RNN

The *Decoder* is a one to many RNN

- Takes the fixed length representation of the seed produced by the Encoder
    - Used to initialized the Decoder's latent state $\mathbf{h}_{(0)}$
- Outputs a variable length sequence

Generative text: state of the art

The model we described (and will explore in a code example) is absolutely primitive

- Predict next *character* rather than next *word*
- Simple: one RNN layer
- Trained on very small number of examples

Here (https://app.inferkit.com/demo) is a link to a state of the art model.

We will learn about its architecture in a later module.

```
In [2]: print("Done")
```

```
Done
```