50.039 – Theory and Practice of Deep Learning

Alex

Week 8: Recurrent Neural Networks 2

3
The program (over 2 to 3 lectures)

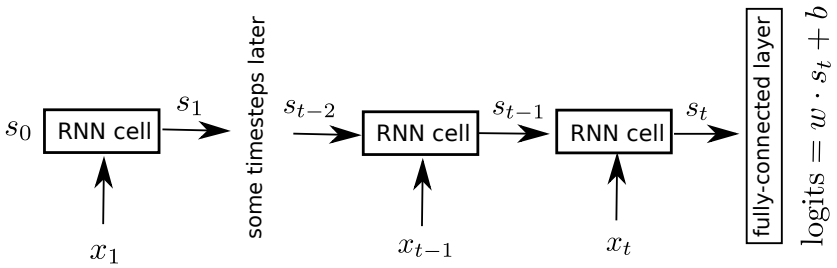
- the problem of vanishing gradients and LSTMs (GRU)
- recurrent neural networks for simple outputs in the sense of computing one single prediction over the whole sequence.
- recurrent neural networks for sequence outputs, sequence to sequence model
- coding: RNNs for language: character level RNNs
- word embeddings
- attention models

Takeaways for today:

- how to use RNNs for learning to generate sequences

1 Code for making a prediction on a sequence as a whole

This is what we want to do:



Here is an example how to implement this with a custom RNN:

`https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html`

Your task will be to replace the custom RNN with an LSTM to get familiar with the usage of an LSTM. There are two options when using the LSTM: feed the whole sequence in one step, or feeding each sequence element step by step in a for loop. The latter is suitable for sequence generation approaches, when one does not know the whole sequence in advance (bcs it is generated). In this example you know the whole sequence in advance, thus one can feed the whole sequence into the LSTM.

To understand how an LSTM is used, see `https://pytorch.org/docs/stable/nn.html#torch.nn.LSTM`, there the LSTM part.
Read the `Inputs` and the `Outputs` part carefully.

It allows to process a whole sequence ... see `Inputs` and the `Outputs`. Alternatively, you pass it in a for loop every element, that is a length-1 sequence.

`https://pytorch.org/docs/stable/nn.html#torch.nn.utils.rnn.pack_sequence` makes sense when one wants to use batchsizes larger than 1 where every element in a batch is a sequence of different sequence length.

2 How to use RNNs for learning to predict on every sequence element

Goal:

- want to predict on every sequence element using the surrounding context
  - Natural Language processing: part of speech tagging - what is a verb, noun, etc in a sentence
  - Genomics: predict which element of a DNA sequence is intro or exon:
    `https://en.wikipedia.org/wiki/Exon`
    `https://www.mun.ca/biology/scarr/Exons_Introns_Codons.html`
- want to predict on every subsequence $(x_0, \ldots, x_t)$ for every t (stock value will go up or down based on x_t = history of news at time step t for all past time steps).

Solution is easy:

- use state vector h_t at every time step t

- train a fully connected output over every h_t with shared weights

$$z_t = w \cdot h_t + b$$

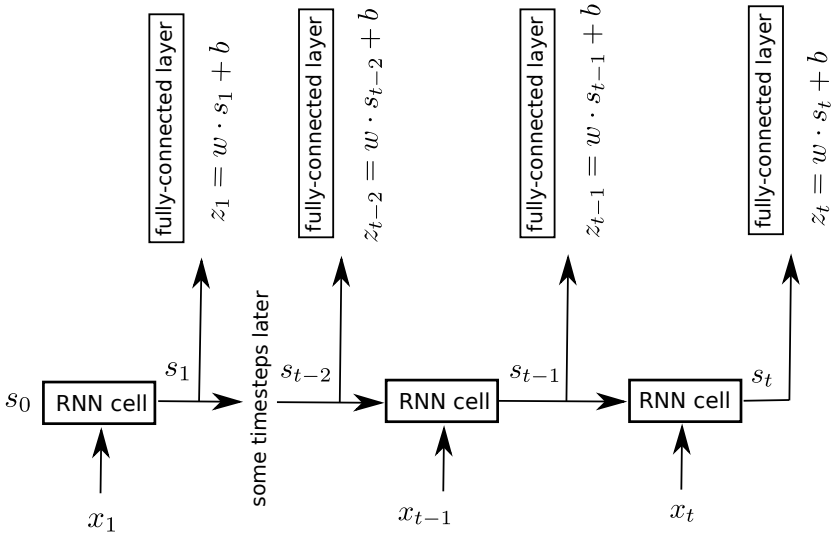
  shared weights means: for every h_t the same w, b is used

- loss will be sum of losses over all timesteps with respect to ground truth y_t:

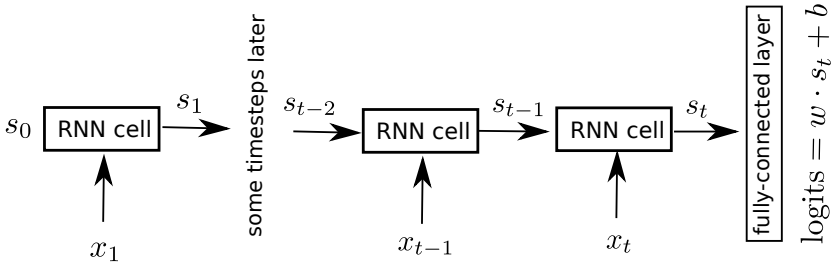
$$L = \frac{1}{T+1} \sum_{t=0}^{T} L(y_t, z_t)$$

Shared weights: pro: less parameters, good if output is structurally same at every times step, con: makes no sense if output behaves differently at every time step

$$L = \frac{1}{t+1} \sum_{r=0}^{t} L(z_r, y_r)$$



compare this to the former setting in which we predict some property on the whole sequence:



Finally note, that this is not restricted to classification. You could attach at every time step instead of the fully connected layer with classification loss a

GAN (generative adversarial neural network) which takes as input s_t / h_t and uses a GAN loss to create some fake face.

A code example is given in `https://pytorch.org/tutorials/beginner/nlp/sequence_models_tutorial.html`. Here again, one knows the whole sequence to be processed in advance.

3 How to use RNNs for learning to generate sequences

Suppose we have a state vector h_0 which contains some aggregated information. For example h_0 = feature activations of a late layer in a convolutional neural network, capturing information about an image. We want to generate a sequence as output from that. How to do that ? Example: image captioning, output should be a sequence of words.

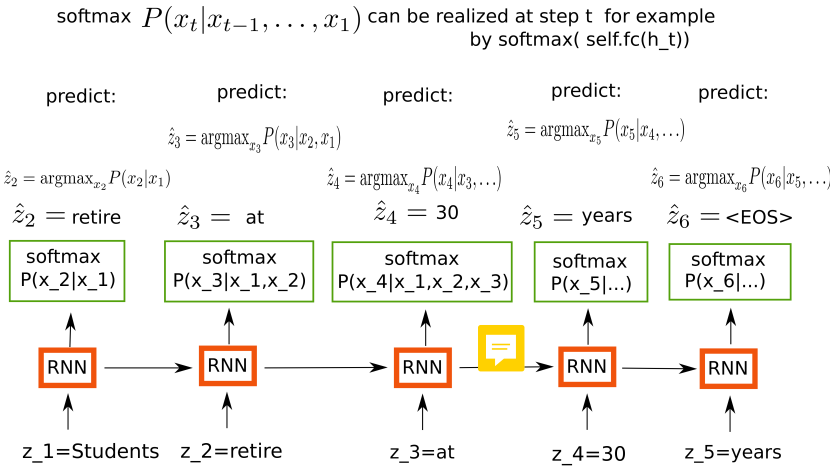
First thing: how to make the network stop outputting ?

- By including a stop symbol in the vocabulary of permitted outputs,
- by appending in the training set the stop symbol at the end of every sequence
- by training it to output the stop symbol (where appropriate)

Second thing: how to make it output the next element ?

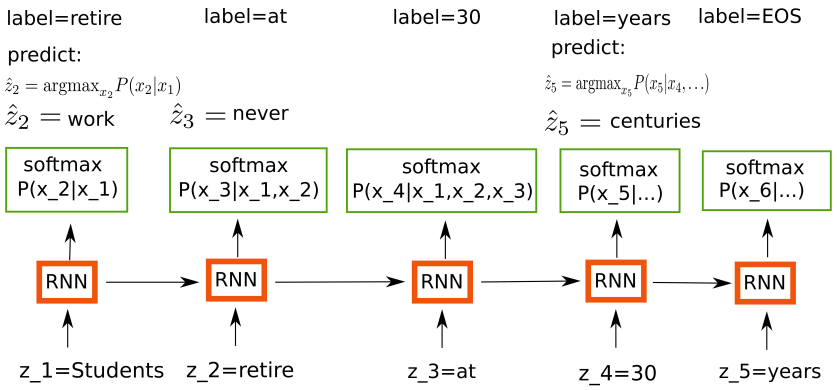
3.1 Prediction: generate a sequence

Idea: feed last prediction z_t as input to the next step!



3.2 Training

The coarse idea is: given a training sequence $z = z_1, \ldots, z_T$, train the RNN to predict z_{t+1} given that it has seen before $z_1, \ldots, z_t$ – and we do this for $t = 1, \ldots, T-1$



We need to specify a model and a loss function (and an EOS token).

- The output model of the softmax at step t is

$$P(x_{t+1}|x_1, \ldots, x_t)$$

- The loss function for a sequence $z_1, \ldots, z_T$ will be:

$$L = \sum_{t=1}^{T-1} loss(P(x_{t+1}|x_1, \ldots, x_t), \text{label}_{t+1} = z_{t+1}) + endterm$$

  That is: $P(x_{t+1}|x_1, \ldots, x_t)$ is a vector of probabilities, one probability value for every possible value which x_{t+1} can take (the label is one of them). It sums up to one over all possible values.

- One example for such as loss can be the cross-entropy loss:

$$loss(P(x_{t+1}|x_1, \ldots, x_t), label = z_{t+1}) = -\log P(x_{t+1} = z_{t+1}|x_1, \ldots, x_t)$$

  Minimizing this loss encourages $P(x_{t+1} = z_{t+1}|x_1, \ldots, x_t) \longrightarrow 1$.

- How do we train the neural network to tell us when it has finished to generate a sequence ?
  We add an **end-of-sequence (EOS) token** to the set of all possible outputs. We want the neural net to output EOS after z_T, that is we like to achieve:

$$EOS = \text{argmax}_z P(x_{T+1} = z|x_1, \ldots, x_T)$$

- We can encourage this behavior by adding an endterm to the loss and training the net.

$$endterm = loss(P(x_{T+1}|x_1, \ldots, x_T), label = EOS)$$

This concludes the idea on how to train a RNN to produce sequences.

For a code example: `https://pytorch.org/tutorials/intermediate/char_rnn_generation_tutorial.html`

Note: in week 6 you will have a homework which will let you train a text generator.

3.3 Better Sampling than $\text{argmax}_{x_t} P(x_t|x_{t-1}, \ldots)$

So far the rule we have used for a softmax output $P(z)$ was meant for choosing one class as classification prediction:

$$y = \text{argmax}_z P(z)$$

This is good for classification. That is a very bad idea when one wants to generate sequences from a model. Why?

For sampling we can use another idea:

- See the probability distribution as probability distribution over likely next sequence elements

- drawing a sequence element z according to the probability distribution $P(z)$ given by the softmax (which here would be the term $P(x_{t+1} = z|x_1, \ldots, x_t)$).

Sampling with a temperature:

There is even a tunable way to sample from the distribution, so that in the limit we can obtain either a uniform distribution, or the distribution which always would select $y = \text{argmax}_z P(z)$. Suppose $P(z)$ is given as a softmax over the vector a (typically the output of a fully connected layer), then we divide it by a temperature C

$$P(z) = softmax(a/C)$$

For $C \to \infty$ we have $a/C \to 0$ and thus $P(z)$ is computed as a softmax over a vector in which every component is zero.

$$\frac{\exp(0)}{\sum_{v=1}^{V} \exp(0)} = \frac{1}{V}$$

is the uniform distribution.

In the other extreme, if $C \to 0$, then we know that in

$$a/C = (a_1/C, \ldots, a_V/C)$$

that the largest term a_k/C will grow faster to ∞ than all the other terms $a_i/C, i \neq k$. Therefore, the softmax of that term will this converge to 1, due to $a_k/C \gg a_i/C \Rightarrow \frac{\exp(a_k/C)}{\sum_{v=1}^{V} \exp(a_v/C)} \gg \frac{\exp(a_i/C)}{\sum_{v=1}^{V} \exp(a_v/C)}$.

The only thing that you need to do is to provide the first letter. Alternatively one can train with an additional SOS (start of sentence) token.

Sampling with a temperature and training

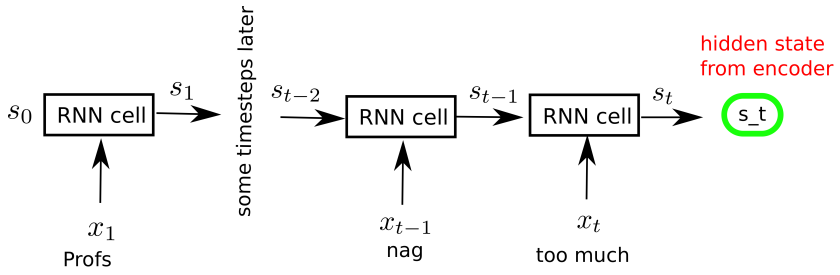
Of course you can use that also to inject more noise in your predictions at training time, so that your algorithm would become more robust. You will need to validate training by choosing a sufficiently cold temperature. $T = 1$ is the default.

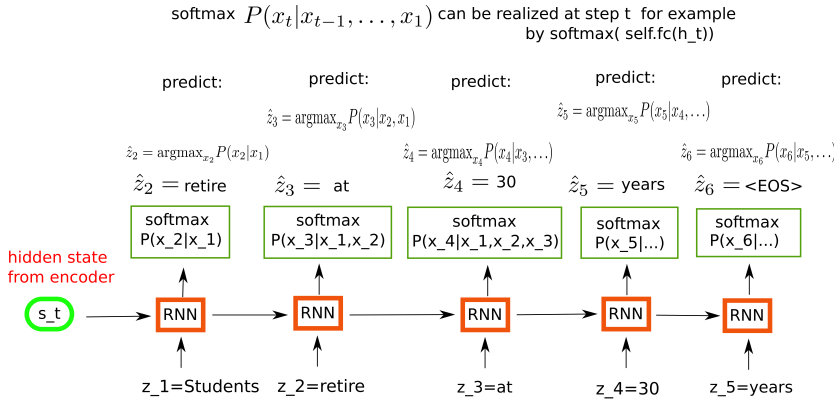
4 Sequence-to-sequence models

Goal: input a sequence, output: another sequence of different and varying length.

combine two things:

- process input sequence x by a RNN, output is a hidden state h
- use hidden state h to initialize a sequence generator RNN
- train with training loss from sequence generation to obtain a meaningful network (see Section How to use RNNs for learning to generate sequences)





Next lectures: word embeddings, attention models

5 Code example on sequence outputs

`https://pytorch.org/tutorials/intermediate/char_rnn_generation_tutorial.html`

Take a look at it:

- first window to read names per category, homework: only one category

- third window `def randomTrainingPair():` random sampling of a name , homework: need a sentence, can solve by iterator again, as in the previous homework
- `randomTrainingExample():` generates an input sentence, and a target sentence with attached **¡EOS¿** flag, puts all into tensors
- second window - the actual model
- `def train(...):` explicitly loops though the rnn, not necessary, need only explicitly sum the loss over all elements of `output`
- test time: `def sample(category, start_letter='A'):`
  - here must for-loop explicitly and feed last element into RNN
  - `break` if one encounters EOS
  - improved sampling would use so called beam search, not a randomized softmax at every step

note: learning to imitate is just one way to instantiate a generative process – possibly it is inferior for complex tasks without proper pretrained embeddings, and it may need some augmentation with some input code providing attributes for topics/meanings, see GANs for an alternative - which turn an input code into an output