50.039 – Theory and Practice of Deep Learning

Alex

Week ??: Recurrent Neural Networks 1

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

Key takeaways:

- the vanishing gradient problem
- the formula how to do backpropagation with respect to a parameter W for one step in a recurrent neural network (from time step E to $E-1$):

$$\frac{ds_E}{dW} = \frac{\partial s_E}{\partial W} + \frac{\partial s_E}{\partial s_{E-1}} \frac{ds_{E-1}}{dW}$$

- a frequently used RNN: LSTM
- how to use RNNs for simple outputs in the sense of computing one single prediction over the whole sequence.

1 Recurrent neural nets

Recurrent neural nets is a tool for

- **processing varying length sequence data** and for
- **generating varying length sequence data** based on neural networks.

Examples are ?

- time series $x = (x_1, x_2, \dots, x_T)$ like ???.
- sentences $x = (x_1 = \text{This}, x_2 = \text{is}, x_3 = \text{a}, x_4 = \text{hungry-looking}, x_5 = \text{fishie}, x_6 = !)$
- counts of passengers in a bus
- the age distribution of the passengers – which may change at every station.

Note: We use the t in x_t as sequence index notation. t does not need to be a time. What is t in the example of the text or the bus counts?
How to process them

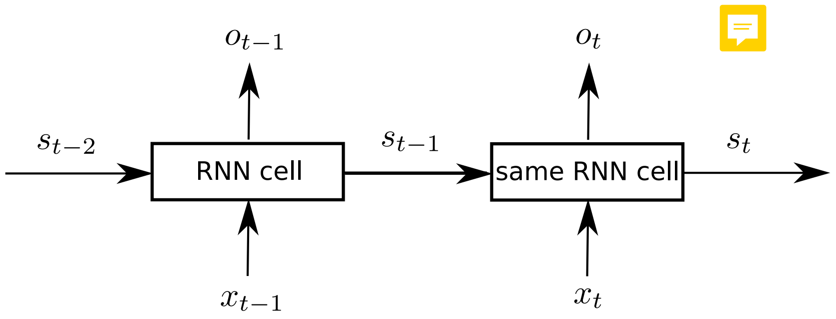
- sequence processing: **input: a sequence.** output: whatever, e.g. class label, a regression output, an image
- sequence generation: input: whatever (image or sequence). **output: another sequence**

Applications of RNNs that you have heard of?

2 RNNs for sequence processing

We want to predict something over a sequence. The prediction can be a number or another sequence. One example is to predict whether to buy, hold or sell at the next time step (prediction would be here 3 numbers - for probabilities), another example is: translate one sentence into another language. Then the prediction would be a sequence – the words of the correct translation. We focus here on non-sequential outputs, the part on sequence processing later.

The first question is: How to use recurrent neural nets to process sequence data? Lets look at one Recurrent neural net cell.



- Recurrent neural net cells have an internal state s_t, often encoded by some vector.

- At step t the element x_t of the sequence x is fed in, and the internal state is updated we compute the new interal state s_t from the old one s_{t-1}, and possibly an output is produce:d: o_t.

$$\begin{aligned} s_t &= f(s_{t-1}, x_t) \\ o_t &= g(x_t, s_t) \\ &\text{or} \\ o_t &= g(x_t, s_{t-1}) \end{aligned}$$

f, g are functions specific to the implementation of recurrent neural nets.

A very simple example is: suppose x_t is a vector in 10 dimensions, the state s_t is a vector in 30 dimensions, the output o_t is a vector in 5 dimensions. Then one could define as equations

$$s_{t-1}.shape = (30,1),\ x_t.shape = (10,1)$$

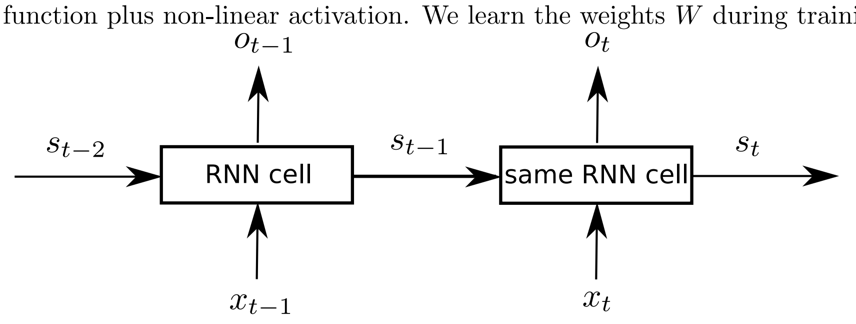
$$s_t = tanh(W_{ss} s_{t-1} + W_{xs} x_t)$$

$$W_{ss}.shape = (30,30), W_{xs}.shape = (30,10)$$

$$o_t = tanh(W_{xo} x_t + W_{so} s_{t-1})$$

$$W_{xo}.shape = (5,10), W_{so}.shape = (5,30)$$

The $tanh$ is applied element-wise – for every component of the vectorial input. The idea here is same as for other neural networks: We apply weights W that can be learned, then apply an activation function on top. The model is a linear function plus non-linear activation. We learn the weights W during training.



The similarity and difference to feedforward networks before:

- You have learnable weight parameters as with convolutional networks,
- you have also a persistent state vector s_t – which depends on the history of inputs and on your learnable parameters.

The state s_t is carried forward from one element in the sequence x_t to another (x_{t+1}).

Famous examples are LSTM (Long Term Short Term memory, Hochreiter and Schmidthuber, 1997), and GRU (gated recurrent unit, Kyunghyun Cho, 2014).

2.1 vanishing/exploding gradients and LSTM

Training recurrent neural nets can suffer from the problem of gradients over time exploding or vanishing when using backpropagation (why historically this was discovered before the gradient problem in convnets ?).
One problem is: consider our simple neuron from above (we dont care for the outputs o_t for a moment)

$$s_t = f_W(s_{t-1}, x_t)$$
$$s_t = tanh(W s_{t-1} + W_{xs} x_t)$$

The parameter W_{ss} is used in every time step from s_0 to s_E. If we want the gradient w.r.t. W_{ss}, then we need to consider the cell state at each of these time steps. Lets write how the hidden state evolves over time:

$$\begin{aligned} s_E &= s_E(W, S_{E-1}) \\ s_{E-1} &= s_{E-1}(W, S_{E-2}) \\ s_{E-2} &= s_{E-2}(W, S_{E-3}) \\ &\vdots \\ s_3 &= s_3(W, S_2) \\ s_2 &= s_2(W, S_1) \\ s_1 &= s_1(W, S_0) \end{aligned}$$

partial versus total differential

What is the partial versus the total differential ?? Suppose a function depends on a parameter W in multiple ways:

$$g(w, u, v)$$
$$g(W, h(W), r(W)) = g(W = W, U = h(W), V = r(W))$$

For the partial differential one considers the derivative only in one variable - usually the direct dependency of the function on this variable.

The partial differential in W computes only the partial derivative with respect to the direct occurrence of W in g, but ignores the indirect dependencies of g on w via the other terms $h(w), r(w)$.

$$\frac{\partial g}{\partial W}(w = w, h(w), g(W)) = \lim_{\epsilon \to 0} \frac{g(w+\epsilon, h(w), r(w)) - g(w-\epsilon, h(w), r(w))}{2\epsilon}$$

The total differential uses chain rule to compute the derivative of g with respect to w in every component, direct and indirect dependencies:

$$\begin{aligned} \frac{dg}{dW}(w, h(w), g(W)) =& \frac{\partial g}{\partial W}(w = w, h(w), g(W)) + ... \\ & \frac{\partial g}{\partial U}(w, u = h(w), g(W)) \frac{\partial h}{\partial w}(w = w) + ... \\ & \frac{\partial g}{\partial V}(w, h(w), v = g(W)) \frac{\partial r}{\partial w}(w = w) \end{aligned}$$

Why is this important here ?

$$\begin{aligned}
s_E &= s_E(W, S_{E-1}) \\
s_{E-1} &= s_{E-1}(W, S_{E-2}) \\
s_{E-2} &= s_{E-2}(W, S_{E-3}) \\
&\vdots \\
s_3 &= s_3(W, S_2) \\
s_2 &= s_2(W, S_1) \\
s_1 &= s_1(W, S_0)
\end{aligned}$$

s_E depends on W directly, and indirectly via $S_{E-1}, S_{E-2}, \ldots$ and so on.

A sloppy explanation of exploding/vanishing gradients:

Lets give at first a sloppy, not really correct, but intuitive derivation: From above you can conclude that S_E is a function of the parameter W acting in the last time step, and of S_{E-1}, S_{E-2}, S_{E-3} and so on:

$$s_E = s_E(W, S_{E-1}(W), S_{E-2}(W), S_{E-3}(W), \ldots, S_2(W), S_1(W))$$

Applying chain rule yields:

$$\frac{ds_E}{dW} = \frac{\partial s_E}{\partial W} + \frac{\partial s_E}{\partial s_{E-1}} \frac{\partial s_{E-1}}{\partial W} + \frac{\partial s_E}{\partial s_{E-2}} \frac{\partial s_{E-2}}{\partial W} + \ldots + \frac{\partial s_E}{\partial s_1} \frac{\partial s_1}{\partial W}$$

one can show (see section Formal derivation) that

$$\begin{aligned}
\frac{\partial s_E}{\partial s_{E-2}} &\to \text{ 2 terms} \\
\frac{\partial s_E}{\partial s_{E-3}} &\to \text{ 3 terms} \\
\frac{\partial s_E}{\partial s_{E-4}} &\to \text{ 4 terms}
\end{aligned}$$

in general it has K terms:

$$\frac{\partial s_E}{\partial s_{E-K}} = \frac{\partial s_E}{\partial s_{E-1}} \frac{\partial s_{E-1}}{\partial s_{E-2}} \frac{\partial s_{E-2}}{\partial s_{E-3}} \frac{\partial s_{E-3}}{\partial s_{E-4}} \cdots \frac{\partial s_{E-K+1}}{\partial s_{E-K}}$$

How can one arrive at this formula? This comes from the fact that:

$$\begin{aligned}
s_E &= s_E(W, S_{E-1}) \\
s_{E-1} &= s_{E-1}(W, S_{E-2}) \\
s_{E-2} &= s_{E-2}(W, S_{E-3}) \\
s_{E-3} &= s_{E-3}(W, S_{E-4})
\end{aligned}$$

lets concatenate these into one chain:

$$\begin{aligned}
&\rightarrow S_E = s_E(W, s_{E-1}(W, s_{E-2}(W, S_{E-3}))) \\
&\rightarrow S_E = s_E(W, s_{E-1}(W, s_{E-2}(W, S_{E-3}(W, s_{E-4}))))
\end{aligned}$$

backtracking this from S_E through s_{E-1} and s_{E-2} until s_{E-4} gives a formula such as

$$\begin{aligned}
\frac{\partial s_E}{\partial s_{E-4}} &= \frac{\partial s_E}{\partial s_{E-1}} \frac{\partial s_{E-1}}{\partial s_{E-4}} \\
&= \frac{\partial s_E}{\partial s_{E-1}} \frac{\partial s_{E-1}}{\partial s_{E-2}} \frac{\partial s_{E-2}}{\partial s_{E-4}} \\
&= \frac{\partial s_E}{\partial s_{E-1}} \frac{\partial s_{E-1}}{\partial s_{E-2}} \frac{\partial s_{E-2}}{\partial s_{E-3}} \frac{\partial s_{E-3}}{\partial s_{E-4}}
\end{aligned}$$

and in general:

$$\frac{\partial s_E}{\partial s_{E-K}} = \frac{\partial s_E}{\partial s_{E-1}} \frac{\partial s_{E-1}}{\partial s_{E-2}} \frac{\partial s_{E-2}}{\partial s_{E-3}} \frac{\partial s_{E-3}}{\partial s_{E-4}} \cdots \frac{\partial s_{E-K+1}}{\partial s_{E-K}}$$

The problem is: products over many terms such as above either diverge to ∞ when absolute values are above 1, or the converge to zero when absolute values are below 1:

$$\begin{aligned}
2^n &\rightarrow \infty \\
1.1^n &\rightarrow \infty \\
0.8^n &\rightarrow 0
\end{aligned}$$

Lets go now for a more correct derivation:

The chain rule for one time step:
Lets write for clarity about gradients the hidden state s_t as a function of the first parameter u - where the variable of interest **appears in the current time step**, and the second parameter v - where the variable of interest **appears in the past time steps**:

$$\begin{aligned}
s_t &= s_t(u, v) \\
u = W, v &= s_{t-1}
\end{aligned}$$

Both input variables u and v depend on W_{ss}, because $s_{t-1} = s_{t-1}(W)$. That means if we start to compute the total derivative, we have to use chain rule and to compute a derivative with respect to the first variable u and with respect to the second variable v:

$$\frac{ds_E}{dW} = \frac{\partial s_E}{\partial u}(u = W, v = s_{E-1})\frac{du}{dW} + \frac{\partial s_E}{\partial v}(u = W, v = s_{E-1})\frac{dv}{dW}$$

Now we now $u = W$, so $\frac{du}{dW} = 1$, and $v = s_{E-1}$, so $\frac{dv}{dW} = \frac{ds_{E-1}}{dW}$. Therefore we obtain:

$$\frac{ds_E}{dW} = \frac{\partial s_E}{\partial u}(u = W, v = s_{E-1}) + \frac{\partial s_E}{\partial v}(u = W, v = s_{E-1})\frac{ds_{E-1}}{dW}$$

You can apply the same now to $\frac{ds_{E-1}}{dW}$:

$$\frac{ds_{E-1}}{dW} = \frac{\partial s_{E-1}}{\partial u}(u = W, v = s_{E-2}) + \frac{\partial s_{E-1}}{\partial v}(u = W, v = s_{E-2})\frac{ds_{E-2}}{dW}$$

and again you can apply the same to $\frac{ds_{E-2}}{dW}$.

Lets use sloppy notation:

$$\frac{\partial s_E}{\partial u}(u = W, v = s_{E-1}) = \frac{\partial s_E}{\partial W}$$
$$\frac{\partial s_E}{\partial v}(u = W, v = s_{E-1}) = \frac{\partial s_E}{\partial s_{E-1}}$$

One step differentiation for recurrent neural nets:

Then we have sloppily written:

$$\frac{ds_E}{dW} = \frac{\partial s_E}{\partial W} + \frac{\partial s_E}{\partial s_{E-1}}\frac{ds_{E-1}}{dW}$$

In words: the total derivative of a RNN cell with respect to a parameter W is the sum of two derivatives:

- the direct, partial derivative of the current RNN cell state s_E with respect to that parameter W: $\frac{\partial s_E}{\partial W}$
- and the derivative of the rnn cell with respect to the previous state $\frac{\partial s_E}{\partial s_{E-1}}$ times the total derivative of that previous state w.r.t to the parameter: $= \frac{\partial s_E}{\partial s_{E-1}}\frac{ds_{E-1}}{dW}$

One can show from here (see formal derivation section ... well formal would be a proof by induction, which is easy):

differentiation for recurrent neural nets completely rolled out

$$\frac{ds_E}{dW} = \frac{\partial s_E}{\partial W} + \frac{\partial s_E}{\partial s_{E-1}}\frac{\partial s_{E-1}}{\partial W} + \frac{\partial s_E}{\partial s_{E-2}}\frac{\partial s_{E-2}}{\partial W} + \ldots + \frac{\partial s_E}{\partial s_1}\frac{\partial s_1}{\partial W}$$

where

- $\frac{\partial s_E}{\partial s_{E-2}}$ is a product of 2 terms
- $\frac{\partial s_E}{\partial s_{E-3}}$ is a product of 3 terms
- $\frac{\partial s_E}{\partial s_{E-4}}$ is a product of 4 terms
- $\frac{\partial s_E}{\partial s_{E-K}}$ is a product of K many terms
- and so on!

$$\frac{\partial s_E}{\partial s_{E-2}} = \frac{\partial s_E}{\partial s_{E-1}}\frac{\partial s_{E-1}}{\partial s_{E-2}}$$
$$\frac{\partial s_E}{\partial s_{E-3}} = \frac{\partial s_E}{\partial s_{E-1}}\frac{\partial s_{E-1}}{\partial s_{E-2}}\frac{\partial s_{E-2}}{\partial s_{E-3}}$$
$$\frac{\partial s_E}{\partial s_{E-4}} = \frac{\partial s_E}{\partial s_{E-1}}\frac{\partial s_{E-1}}{\partial s_{E-2}}\frac{\partial s_{E-2}}{\partial s_{E-3}}\frac{\partial s_{E-3}}{\partial s_{E-4}}$$

$$\frac{\partial s_E}{\partial s_{E-K}} = \frac{\partial s_E}{\partial s_{E-1}}\frac{\partial s_{E-1}}{\partial s_{E-2}}\frac{\partial s_{E-2}}{\partial s_{E-3}}\frac{\partial s_{E-3}}{\partial s_{E-4}}\cdots\frac{\partial s_{E-K+1}}{\partial s_{E-K}}$$

What is the problem ? For computing the gradient over K time steps, we will need to have a product over $O(K)$ terms. Product of many terms tend either to explode or to shrink to zero! This is the vanishing/exploding gradient problem

2.2 LSTM

LSTM was designed to be more robust against it. It has a hidden state s and a memory cell c. For the memory cell $dc_t/dc_{t-1} \approx 1$.

Note: I am not requiring you to memorize LSTM equations, but rather the concepts of having a memory cell, input gates to update the memory cell (and forget gates) ,and of above gradient property about robustness to the vanishing gradient problem.

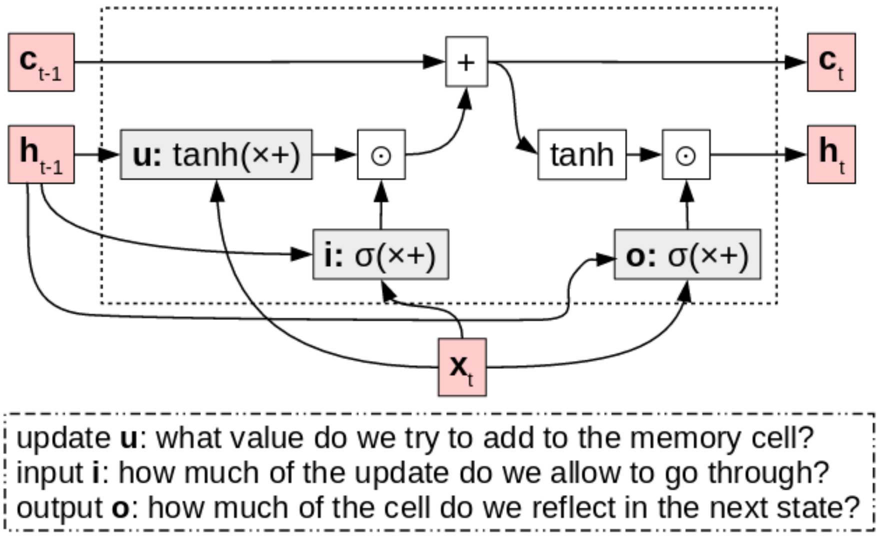

c_{t-1}
h_{t-1}
u: tanh(×+)
⊙
+
tanh
⊙
c_t
h_t
i: σ(×+)
o: σ(×+)
x_t
update **u**: what value do we try to add to the memory cell?
input **i**: how much of the update do we allow to go through?
output **o**: how much of the cell do we reflect in the next state?

Image credit: Graham Neubigs Tutorial (see below)

From a high level perspective:

LSTM explained:

- in an LSTM there are two parts, which are kept through time: the hidden state h_t and the memory cell c_t.
- compute an update u_t for the memory cell:
  $u_t = u(h_{t-1}, x_t)$
- compute an input weight i_t for the update u_t for the memory cell:
  $i_t = i(h_{t-1}, x_t)$
- update memory cell by **weighted** update:
  $c_t = c_{t-1} + i_t \odot u_t$
  ($\odot$: element-wise multiplication, no matrix multiplication)
- compute output weight, that decides how much from the memory cell will be reflected in the next hidden state:
  $o_t = o(h_{t-1}, x_t)$
- compute next hidden state as a product of output weight and *tanh* of the memory cell: $h_t = o_t \odot tanh(c_t)$

LSTM equations:

Lets look at the formulas for $i(\cdot), u(\cdot), o(\cdot)$ and see that they are not so confusing on the second sight:

$$\begin{aligned}
\boldsymbol{u_t} &= u(\boldsymbol{x_t}, \boldsymbol{h_{t-1}}) && = tanh(W_{x,u}\boldsymbol{x_t} + W_{h,u}\boldsymbol{h_{t-1}} + \boldsymbol{b_u}) \\
\boldsymbol{i_t} &= i(\boldsymbol{x_t}, \boldsymbol{h_{t-1}}) && = \sigma(W_{x,i}\boldsymbol{x_t} + W_{h,i}\boldsymbol{h_{t-1}} + \boldsymbol{b_i}) \\
& \boldsymbol{c_t} = \boldsymbol{c_{t-1}} + \boldsymbol{i_t} \odot \boldsymbol{u_t} \\
\boldsymbol{o_t} &= o(\boldsymbol{x_t}, \boldsymbol{h_{t-1}}) && = \sigma(W_{x,o}\boldsymbol{x_t} + W_{h,o}\boldsymbol{h_{t-1}} + \boldsymbol{b_o}) \\
& \boldsymbol{h_t} = \boldsymbol{o_t} \odot tanh(\boldsymbol{c_t})
\end{aligned}$$

a top down look on LSTM equations:

- the functions $i(\cdot), u(\cdot), o(\cdot)$ are always realized as:
  - linear sums of (x_t, h_{t-1}) with weights $[W_x, W_h]$

    $$W_{x,(\cdot)}x_t + W_{h,(\cdot)}h_{t-1}, \ (\cdot) = i, o, u$$

  - on top of that an activation function $(tanh, \sigma)$
- When to use $\sigma(\cdot)$ and when $tanh(\cdot)$?
  - weights are computed using sigmoids to be in $[0, 1]$.
  - Updates into the memory cell and from it are computed using $tanh$ to be in $[-1, +1]$.

- one can add a forget gate to allow the memory cell to clear its contents partially. the equation $c_t = i_t \odot u_t + c_{t-1}$. changes to $c_t = i_t \odot u_t + f_t \odot c_{t-1}$. where f_t is a vector of forget weights.
- memory may exist on different time-scales (like when reading a book. Information can be relevant from previous chapters and from the previous paragraph.)
- alternatives/ further extension: gated recurrent unit (GRU), bidirectional recurrent neural networks, models with attention modeling.

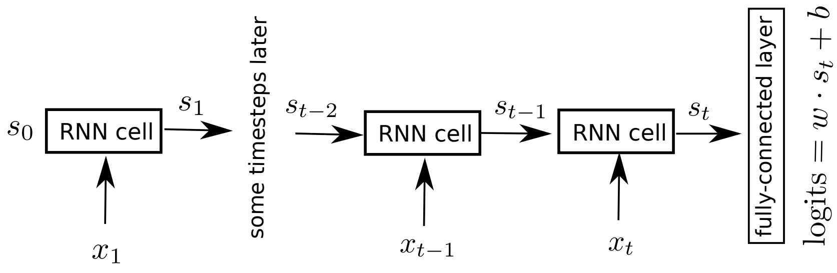
Why does LSTM helps to ease the vanishing gradient problem?

- $\frac{\partial c_t}{\partial c_{t-1}} \approx 1$ – stable gradient flow
- $\boldsymbol{c_t} = \boldsymbol{c_{t-1}} + \boldsymbol{i_t} \odot \boldsymbol{u_t}$ – Residual Connections in the year 1999. (Ask yourself: why people did not apply this to convnets in the year 2000 ?)

3 How to use for producing simple outputs over a sequence?

- for every input sample re-initialize the hidden state to the same constant value (e.g. zeros!)

- Feed your inputs $x_1, \ldots, x_E$ into the RNN/ LSTM, every time update the hidden state $s_1, s_2, \ldots, s_E$.
- take the hidden state after the last step s_E as a feature vector for input for further processing
  - example: want to classify ? Then use the hidden state after the last step s_E as input to a fully connected layer. Done! This works for any length of sequence (in practice it may be useful to train different classifiers for different histogram bins/intervals of sequence lengths)



- interpretation:
  - hidden state s_t aggregates information from the input sequence over time
  - training of the whole pipeline (LSTM with – for example – attached classifier layer) makes the LSTM learn what to extract from the sequence elements $x_1, x_2, x_3, \ldots,$ into the hidden state s_t for the purpose of a high accuracy in classification (learns to extract so that to minimize whatever loss function you define).

4 Code example

`https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html`
Lets go through the code

- lots of code necessary to prepare the data `def lineToTensor(line):` returns a $(len, 1, \#alphabet)$ tensor, which is a sequence of $(1, 1, \#alphabet)$ one-hot encodings with length *len*
- uses random sampling to return a sample from the dataset `def randomTrainingExample():`
- it uses a custom RNN
- it does not use an optimizer to optimize parameters
- training: it explicitly loops through the sequence (why i find that not the best idea?)