Lab 10 : Recurrent Neural Networks (RNNs)

Jeroen Olieslagers
Richard-John Lin

Center for Data Science

11/29/2022

Viterbi

- Use the log formulation of the expression for better stability

$$\begin{aligned} \log v_1(j) &= \log \pi_j + \log C_j(x_1) \\ \log v_n(j) &= \log C_j(x_n) + \max_{i \in [\![1,K]\!]} \log v_{n-1}(i) + \log A_{ij} \\ b_n(j) &= \arg\max_{i \in [\![1,K]\!]} \log v_{n-1}(i) + \log A_{ij} + \log C_j(x_n). \end{aligned}$$

- When in doubt about matrices orientation, look at the sum over the axis. Since we have probabilities, it has to sum to 1 in a direction.
- Watch out for indices when reversing the path.

Sampling

- Whether the initial state has an observation is up to conventions.
- We cannot just randomly sample a state and an observation; it has to take into account the state we are standing in.
- Make sure that the latent state and the observation align; e.g. you can check that a punctuation gives a punctuation in this specific case.

Motivation

- Not all problems can be constrained as a fixed length input and output.
- Especially in time series, we need to store context information, with varying range.
- RNNs aims to approximate the transition function. The intermediate values are supposed to store information about past inputs.
- This is possible because of the shared the weights across time steps.

Principle

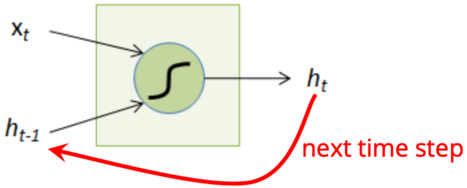


Figure 1: A recurrent neuron

- x_t : input at time t
- h_{t-1} : state at time $t-1$
- $h_t = f(W_j h_{t-1} + W_x x_t)$.

Unfolding the network

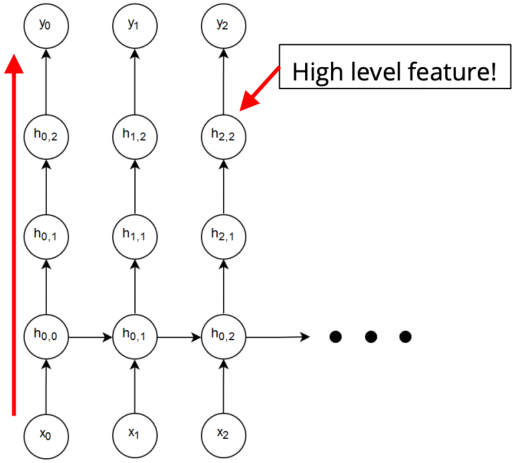


Figure 2: Feedforward depth

Unfolding the network

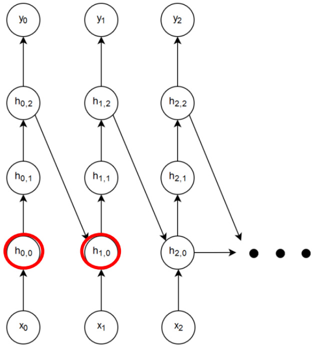


Figure 2: Recurrent depth

- Those are a bit more expressive in theory as they use higher level features.

Input output cases

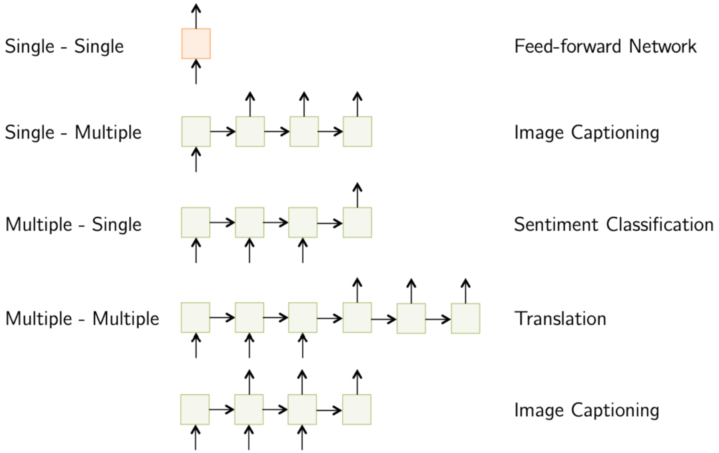


Figure 3: Input output cases

One to many : Image captioning

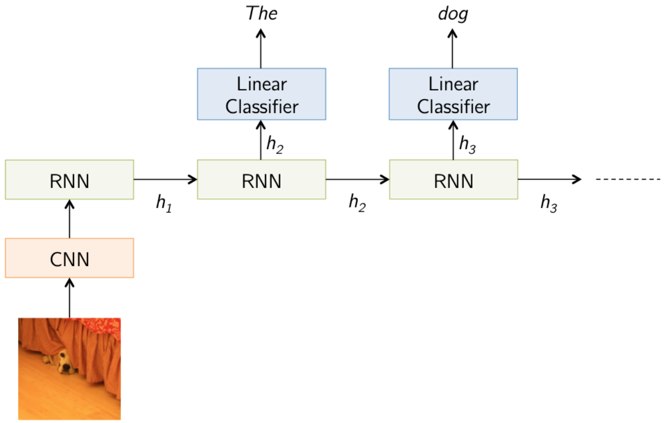


Figure 4: Image captioning

Many to one : Sentiment classification

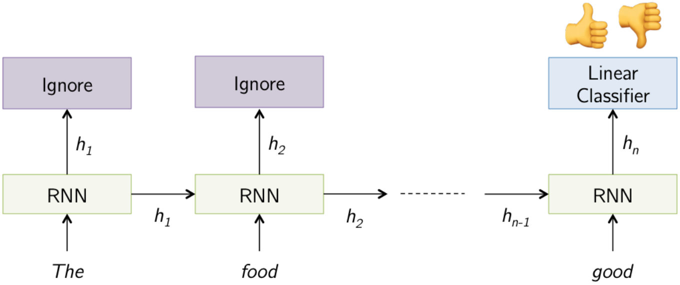


Figure 5: Sentiment classification

Many to one : Sentiment classification

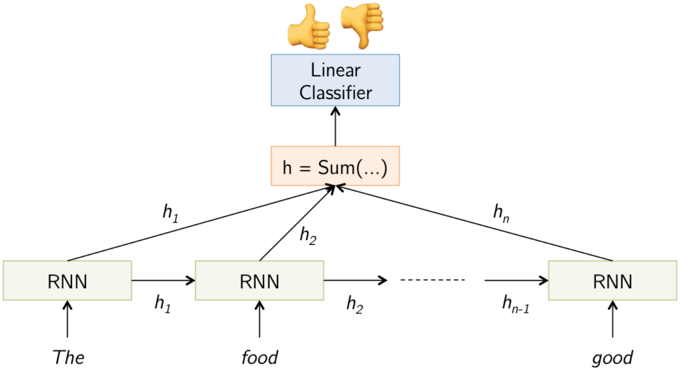


Figure 5: Sentiment classification

Backpropagation Through Time

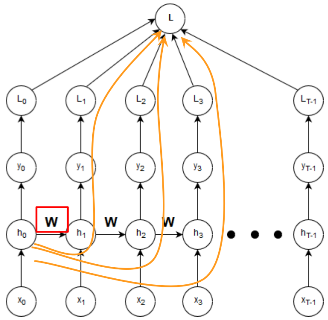


Figure 6: Loss

- The aim is to update the weights

$$W_h \to W_h - \eta \frac{\partial L}{\partial W_h}.$$

- We have :

$$\frac{\partial L}{\partial W_h} = \sum_{j=0}^{T-1} \frac{\partial L_j}{\partial W_h}.$$

Using the chain rule

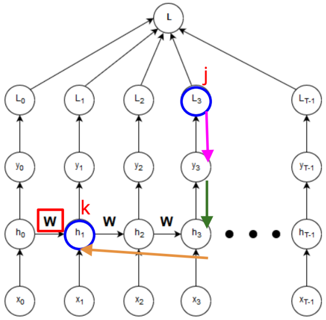


Figure 7: Chain rule

$$\frac{\partial L_j}{\partial W_h} = \sum_{k=1}^{j} \frac{\partial L_j}{\partial h_k} \frac{\partial h_k}{\partial W_h}$$

$$\frac{\partial L_j}{\partial h_k} = \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \frac{\partial h_j}{\partial h_k}$$

$$\frac{\partial h_j}{\partial h_k} = \prod_{m=k+1}^{j} \frac{\partial h_m}{\partial h_{m-1}}$$

BPTT

$$\frac{\partial L}{\partial W_h} = \sum_{j=0}^{T-1} \sum_{k=1}^{j} \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \left(\prod_{m=k+1}^{j} \frac{\partial h_m}{\partial h_{m-1}} \right) \frac{\partial h_k}{\partial W_h}$$

$$h_m = f(W_h h_{m-1} + W_x x_m)$$

$$\frac{\partial h_m}{\partial h_{m-1}} = W_h^\top \mathrm{diag}\left(f'\left(W_h h_{m-1} + W_x x_m\right)\right).$$

In practice, we limit the number of terms in the backpropagation : truncated BPTT.

Vanishing gradient

- Unfortunately, the repeated matrix multiplications in $\frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_k}$ cause optimisation issues.
- Exploding gradients are less a problem, because it is always possible to clip it.
- For vanishing gradient, there are many proposed solutions, including
  - Better weight initialisation methods
  - Constant Error Carousel (LSTM, GRU)

Weight initialisation methods

- If W_h is random, there are no constraints on the eigenvalues, so exploding / vanishing gradients in the initial epoch
- One possible trick is to initialise W_h as positive definite, having at least one eigenvalue 1, and the rest smaller than 1.

Constant Error Carousel

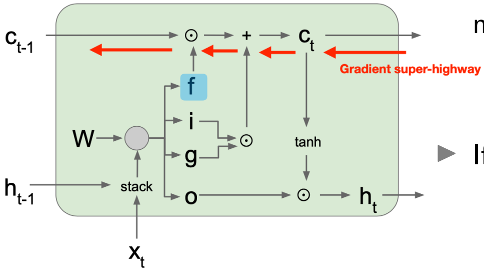


Figure 8: LSTM

- Gradients problem are mitigated by this equation :

$$c_t = f \odot c_{t-1} + i \odot g.$$

- If f is almost one, :

$$\frac{\partial h_t}{\partial h_{t-1}} = 1$$

  No gradient decay.

- $i \odot g$ is similar to a skip connection in resnets

References

- Slides from Arun Mallya (link)
- Slides from Abishek Narwekar and Anusri Pampari (link)

Part I: RNN update rules

Below is our "RNN" class. An instance of RNN is used to run the process *forwards*, keeping track of relevant variables in the process.

$$\mathbf{h}^{(t)} = \mathbf{W}^{\text{rec}}\mathbf{a}^{(t-1)} + \mathbf{W}^{\text{in}}\mathbf{x}^{(t)} + \mathbf{b}^{\text{rec}}$$
$$\mathbf{a}^{(t)} = \phi(\mathbf{h}^{(t)})$$

It also projects to an output space via

$$\mathbf{z}^{(t)} = \mathbf{W}^{\text{out}}\mathbf{a}^{(t)} + \mathbf{b}^{\text{out}}.$$

We call these z values the "pre-outputs," since they are not themselves the actual prediction, but rather that which is passed to some function ψ, e.g. sigmoid or softmax into a space of classes.(Think logits vs. actual classification probabilities.) We denote the prediction

$$\mathbf{y}_{\text{hat}}^{(t)} = \psi(\mathbf{z}^{(t)})$$

as "y_hat" in the code. (For the purpose of this lab, we will use the identity as ψ for convenience.) Then we finally calculate the loss, but we actually do so as a function of $\mathbf{z}$ (and the training label $\mathbf{y}^{*(t)}$), with the output included in the loss function. This is because it's easier for certain types of losses with singularities, such as sigmoid- or softmax-cross-entropy.

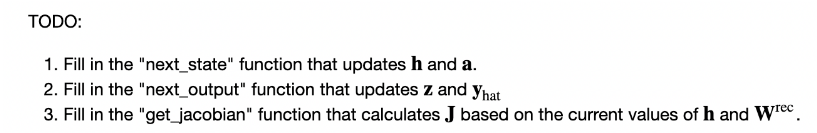
$$L^{(t)} = L(\mathbf{z}^{(t)}, \mathbf{y}^{*(t)})$$

This loss *value* is denoted by "loss_", and the function that calculates it is "loss".

For the purpose of learning, we must also calculate the "Jacobian" of the network, which we include in the RNN class because the mathematical form of the Jacobian varies by RNN architecture---remember, these update equations are just one case of a more general $F_{\mathbf{w}}(\mathbf{a}^{(t-1)}, \mathbf{x}^{(t)})$.

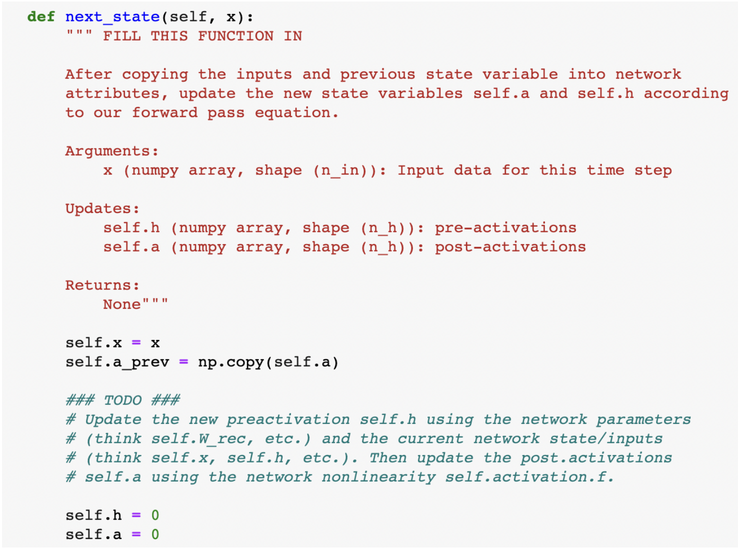
$$J_{ij}^{(t)} = \partial a_i^{(t)}/\partial a_j^{(t-1)} = \phi'(h_i^{(t)})W_{ij}^{\text{rec}}$$

Part I: RNN update rules


TODO:

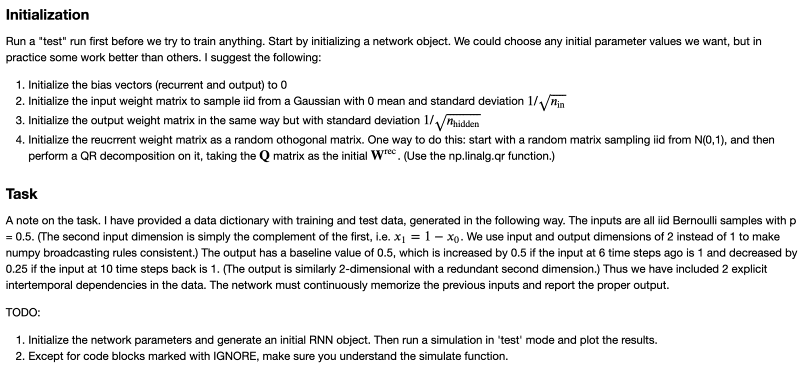
1. Fill in the "next_state" function that updates $\mathbf{h}$ and $\mathbf{a}$.
2. Fill in the "next_output" function that updates $\mathbf{z}$ and $\mathbf{y}_{\text{hat}}$
3. Fill in the "get_jacobian" function that calculates $\mathbf{J}$ based on the current values of $\mathbf{h}$ and $\mathbf{W}^{\text{rec}}$.

Part I: RNN update rules



```python
def next_state(self, x):
    """ FILL THIS FUNCTION IN

    After copying the inputs and previous state variable into network
    attributes, update the new state variables self.a and self.h according
    to our forward pass equation.

    Arguments:
        x (numpy array, shape (n_in)): Input data for this time step

    Updates:
        self.h (numpy array, shape (n_h)): pre-activations
        self.a (numpy array, shape (n_h)): post-activations

    Returns:
        None"""

    self.x = x
    self.a_prev = np.copy(self.a)

    ### TODO ###
    # Update the new preactivation self.h using the network parameters
    # (think self.W_rec, etc.) and the current network state/inputs
    # (think self.x, self.h, etc.). Then update the post.activations
    # self.a using the network nonlinearity self.activation.f.

    self.h = 0
    self.a = 0
```

Part II: Initialization



Initialization

Run a "test" run first before we try to train anything. Start by initializing a network object. We could choose any initial parameter values we want, but in practice some work better than others. I suggest the following:

1. Initialize the bias vectors (recurrent and output) to 0
2. Initialize the input weight matrix to sample iid from a Gaussian with 0 mean and standard deviation $1/\sqrt{n_{\text{in}}}$
3. Initialize the output weight matrix in the same way but with standard deviation $1/\sqrt{n_{\text{hidden}}}$
4. Initialize the reucrrent weight matrix as a random othogonal matrix. One way to do this: start with a random matrix sampling iid from N(0,1), and then perform a QR decomposition on it, taking the $\mathbf{Q}$ matrix as the initial $\mathbf{W}^{\text{rec}}$. (Use the np.linalg.qr function.)

Task

A note on the task. I have provided a data dictionary with training and test data, generated in the following way. The inputs are all iid Bernoulli samples with p = 0.5. (The second input dimension is simply the complement of the first, i.e. $x_1 = 1 - x_0$. We use input and output dimensions of 2 instead of 1 to make numpy broadcasting rules consistent.) The output has a baseline value of 0.5, which is increased by 0.5 if the input at 6 time steps ago is 1 and decreased by 0.25 if the input at 10 time steps back is 1. (The output is similarly 2-dimensional with a redundant second dimension.) Thus we have included 2 explicit intertemporal dependencies in the data. The network must continuously memorize the previous inputs and report the proper output.

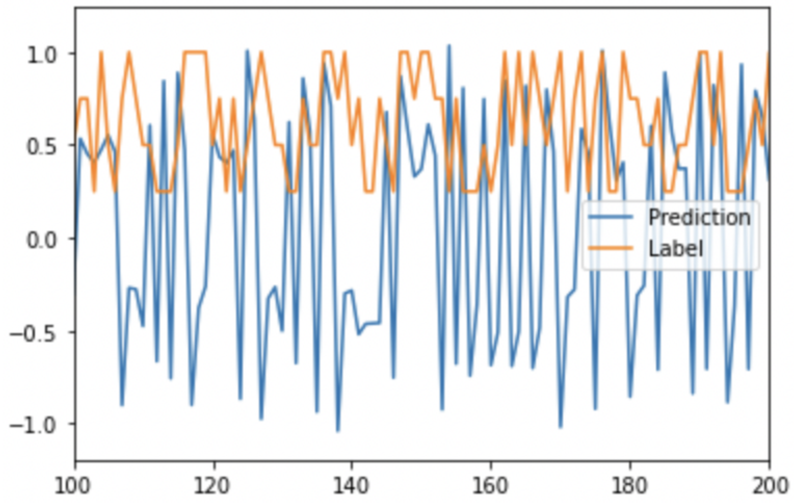
TODO:

1. Initialize the network parameters and generate an initial RNN object. Then run a simulation in 'test' mode and plot the results.
2. Except for code blocks marked with IGNORE, make sure you understand the simulate function.

Part II: Initialization

```
n_in = 2
n_hidden = 32
n_out = 2

np.random.seed(0)

""" FILL IN PARAMETER INITIALIZATIONS """
#Initialize input/output weights as
W_in  = #shape = (n_hidden, n_in))
W_out = #shape = (n_out, n_hidden))

#Initialize recurrent weights with a random *orthogonal* matrix
W_rec = #shape = (n_hidden, n_hidden))

#Initialize biases to 0
b_rec = #shape = (n_hidden)
b_out = #shape = (n_out)

#Initialize RNN object with these initial weights, \phi = tanh, \psi = I, L = MSE
rnn = RNN(W_in, W_rec, W_out, b_rec, b_out,
         activation=tanh,
         output=identity,
         loss=mean_squared_error)

rnn, mons = simulate(rnn, data)
```

Part II: Initialization



Part III: Learning rule (BPTT)

Of course, this untrained network is useless. So let's try training by BPTT, using the "credit assignment vector"

$$\mathbf{q}^{(t)} \equiv \partial \mathcal{L} / \partial \mathbf{a}^{(t)}.$$

We can unpack this expression going backwards via

$$\mathbf{c}^{(t'-1)} = \mathbf{q}^{(t'-1)} + \mathbf{c}^{(t')} \mathbf{J}^{(t')}$$

for any t' in the range from $t - T$ to t, where T is the trunction horizon. We start with $\mathbf{c}^{(t)} = \mathbf{q}^{(t)}$, calculate each of the $\mathbf{c}$ values going backwards, and then get an estimate of the gradient at each time step t'

$$\begin{aligned} \partial \mathcal{L} / \partial W_{ij}^{(t')} &= \mathbf{c}_i^{(t')} \partial a_i^{(t')} / \partial W_{ij}^{(t')} \\ &= \mathbf{c}_i^{(t')} \phi'(h_i^{(t)}) \hat{a}_j^{(t-1)} \end{aligned}$$

We finally sum these up and return it as the gradient.

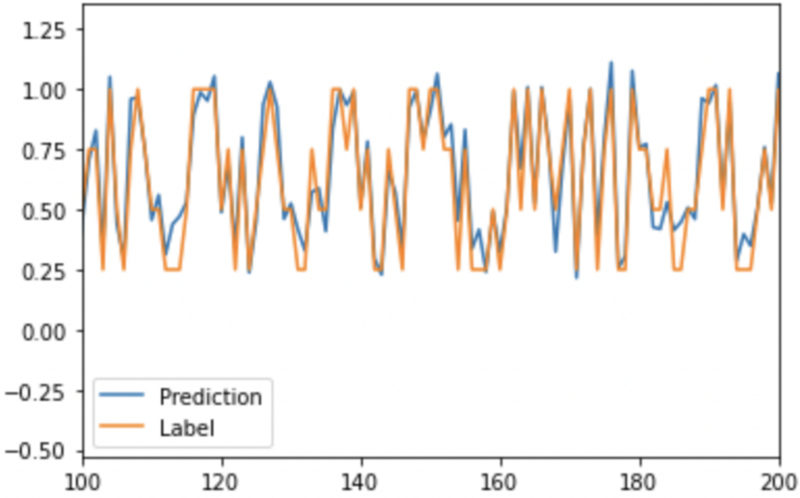
TODO

1. Update the $\mathbf{c}$s going backwards in the get_rec_grads method.

Part III: Learning rule (BPTT)

```
def get_rec_grads(self):

    """ FILL IN FUNCTION

    Using the accumulated history of q, h and a_hat values,
    calculate the gradient of W

    Returns:
        rec_grads (numpy array, (n_h, n_h + n_in + 1)): Gradient \partial L / \partial W as computed
            by backpropagating through the unrolled graph.
    """

    #Hint: initialize the desired output to 0
    rec_grads = 0
    #Hint: initialize first c value to the last q value
    c = self.q_history[-1]
    for i_BPTT in range(self.T_truncation):

        #Access present values of h and a_hat
        h = self.h_history[-(i_BPTT + 1)]
        a_hat = self.a_hat_history[-(i_BPTT + 1)]

        #Get immediate influence
        D = self.rnn.activation.f_prime(h)
        M = np.multiply.outer(D, a_hat)

        rec_grads += (c * M.T).T

        if i_BPTT == self.T_truncation - 1:
            continue

        q = self.q_history[-(i_BPTT + 2)]
        J = self.rnn.get_jacobian(h=h)
        # TODO: update cs
        c = c
```

Part III: Learning rule (BPTT)


Prediction
Label