Neural Network for Synthesizing DFAs

TOC Project - Presentation 02

Abhinav Nakarmi & Kuber Shahi

24th Apr’21

Problem Statement (Paper)

1. **Input:**
    - a Regular Language L to generate the list of tuples (w, *ans*) for training where *ans* = 1 if w $\in$ L , and ans = 0 if w $\notin$ L.
2. **Process:**
    - Train the labelled data (list of tuples) with the proposed RNN
    - For each example (tuples) in the list, the proposed RNN
        i. processes the word character-by-character.
        ii. learns to predict the correct output through stochastic gradient descent and backpropagation.
3. **Output:**
    - DFA *M* such that L(*M*) = L

Proposed Network Architecture

- Based on RNN, contains two parts: Recurrent part, and the adder

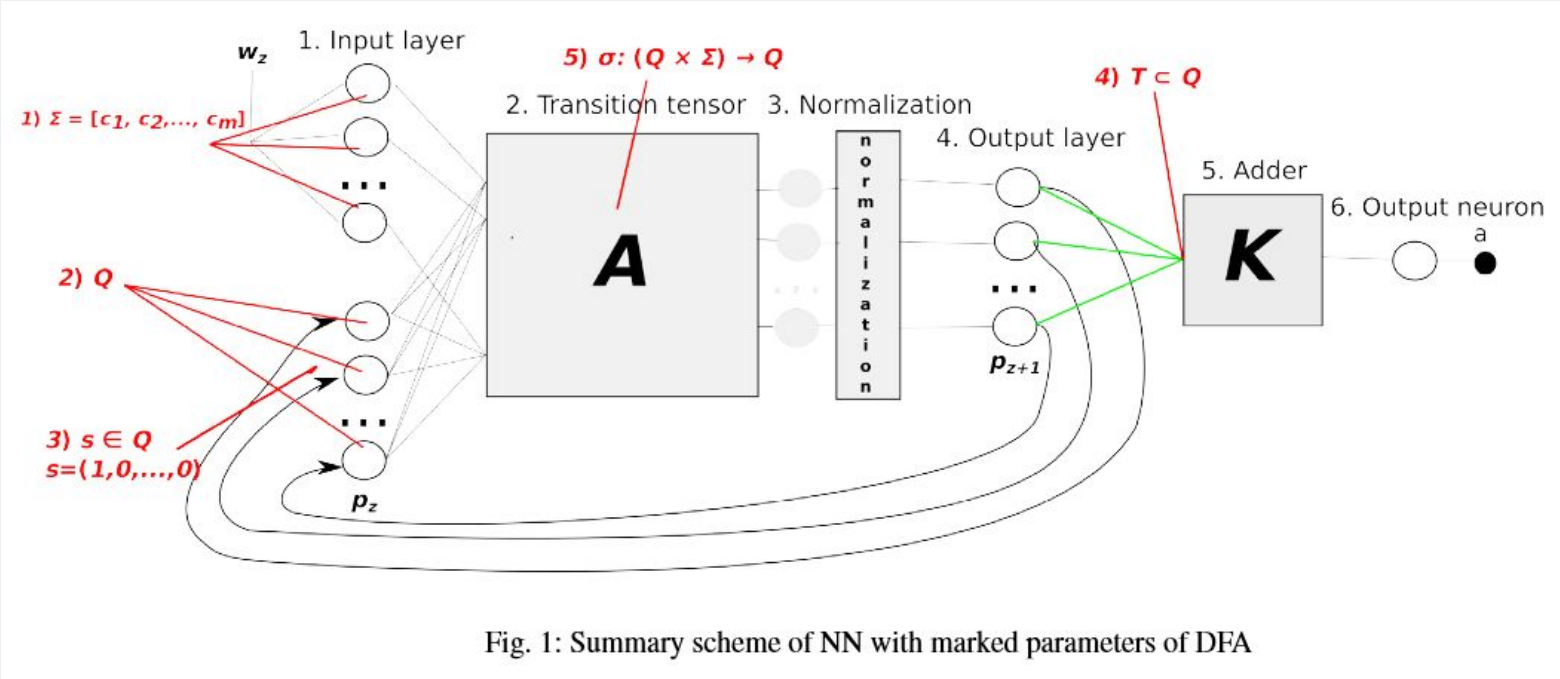


Fig. 1: Summary scheme of NN with marked parameters of DFA

Source: Paper

Training the Recurrent Neural Network

- The proposed RNN mimics two things of a DFA:
  - the transition tensor *A* simulates the transition function (δ).
  - and, the adder vector k simulated the set of accepting states (F).
- Hence, we train the proposed RNN on the labelled data (list of tuples) such that tensors *A* and adder k mimics the δ and F respectively.
- To train the RNN, we will use stochastic gradient descent with backpropagation to minimize the loss.
- The training will be stopped once an loss(cost) is lesser than an specified cost. (we want loss of the proposed RNN to be < 0.01)
- Now, let’s see how gradient descent and backpropagation algorithm works in the case of proposed RNN.

Gradient descent for *A* and *K*

- The transition tensor *A* will be updated as:

$$A_{kji}(new) = A_{kji}(old) - \eta \times \frac{\partial L}{\partial A}$$

- Similarly, the adder vector *k* will be updated as:

$$k_i(new) = k_i(old) - \eta \times \frac{\partial L}{\partial k}$$

Computing gradients of function is the RNN

- Derivative of adder with respect to parametric vector k, where k_i can be interpreted as the probability of that i-th state is terminal.

$$\frac{\partial Adder_k(o)}{\partial k} = \frac{< k, o >}{\partial k} = o$$

Where o is output vector from normalization

- Derivative of adder with respect to output vector, where o_i can be interpreted as the probability that the transition using the next character.

$$\frac{\partial Adder_k(o)}{\partial o} = \frac{< k, o >}{\partial o} = k$$

Where k is parametric vector

Normalize the output layer

In this layer, $\mathbf{h_k}$ is normalized giving output vector $\mathbf{o_i}$ which is calculated as:

$$o_i = \frac{h_i}{\sum_k h_k}$$

- Where, o_i is the output signal value of recurrent part
- h is output of transition function Match

$$\frac{\partial Normalize_i(h)}{\partial h_k} = \frac{\partial \frac{h_i}{<h_i,1>}}{\partial h_k} = \begin{cases} \frac{<h_i,1>-h_i}{<h_i,1>^2} & i = k \\ \frac{-h_i}{<h_i,1>^2} & else \end{cases}$$

- Compute the value of the vector to which the function was applied.
- The value(h) is the result of prior calculations of the transition function Match.

$$\frac{\partial Normalize_i(h)}{\partial h_k} = \frac{\partial \frac{h_i}{<h_i,1>}}{\partial h_k} = \begin{cases} \frac{<h_i,1> - h_i}{<h_i,1>^2} & i = k \\ \frac{-h_i}{<h_i,1>^2} & else \end{cases}$$

o_i= [A B C]

h_j = [X Y Z]

$$\frac{\partial \frac{o_i}{<o_i,1>}}{\partial h_j} = \begin{bmatrix} \frac{\partial A}{\partial X} & \frac{\partial B}{\partial X} & \frac{\partial C}{\partial X} \\ \frac{\partial A}{\partial Y} & \frac{\partial B}{\partial Y} & \frac{\partial C}{\partial Y} \\ \frac{\partial A}{\partial Z} & \frac{\partial B}{\partial Z} & \frac{\partial C}{\partial Z} \end{bmatrix}$$

Transition function Match.

- Derivative of Match with respect to state o, which represents the current state.

$$\frac{\partial Match_k(o_t, w)}{\partial o_t} = \frac{\partial \sum_{i=1}^{|Q|} \sum_{j=1}^{|C|} A_{(k,j,i)} \cdot o_{ti} \cdot w_j}{\partial o_t} = \sum_{j=1}^{|C|} A_{(k,j,t)} \cdot w_j$$

$$\frac{\partial Match_k(o_t, w)}{\partial o_t} = \frac{\partial o_{t+1}}{\partial o_t}$$

o_{t+1}= [A B C]

o_t = [X Y Z]

$$\frac{\partial o_{t+1}}{\partial o_t} = \begin{bmatrix} \frac{\partial A}{\partial X} & \frac{\partial B}{\partial X} & \frac{\partial C}{\partial X} \\ \frac{\partial A}{\partial Y} & \frac{\partial B}{\partial Y} & \frac{\partial C}{\partial Y} \\ \frac{\partial A}{\partial Z} & \frac{\partial B}{\partial Z} & \frac{\partial C}{\partial Z} \end{bmatrix}$$

- Derivative of Match with respect to transition tensor A.

$$\frac{\partial Match_k(o_t, w)}{\partial A_{(k,j,i)}} = \frac{\partial \sum_{i=1}^{|Q|} \sum_{j=1}^{|C|} A_{(k,j,i)} \cdot o_{ti} \cdot w_j}{\partial A_{(k,j,i)}} = o_{ti} \cdot w_j$$

$$\frac{\partial Match_k(o_t, w)}{\partial A_{(k,j,i)}} = \frac{\partial o_{t+1}}{\partial A_{(k,j,i)}} = o_{ti} \cdot w_j$$

o_{t+1} = [A B] which represents the next state

w_j = [X Y] which represents the character

$$\frac{\partial o_{t+1}}{\partial A_{(k,j,i)}} = \begin{bmatrix} \frac{\partial A}{\partial A_{(k,1,1)}} & \frac{\partial B}{\partial A_{(k,1,2)}} \\ \frac{\partial A}{\partial A_{(k,2,1)}} & \frac{\partial B}{\partial A_{(k,2,2)}} \end{bmatrix}$$

Backpropagation: computing gradient for k

- The loss function is $L(y(w), RNN(w)) = (y(w) - RNN(w))^2$
- While updating adder weights *k*, we use:

$$k_i(new) = k_i(old) - \eta \times \frac{\partial L}{\partial k}$$

- We compute the derivative term using back propagation:

$$\frac{\partial L}{\partial k} = 2 \times [y(w) - RNN(w)] \times (-1) \times \frac{\partial RNN(w)}{\partial k}$$

$$\frac{\partial L}{\partial k} = 2 \times [y(w) - RNN(w)] \times (-1) \times \frac{\partial Adder(o_w)}{\partial k}$$

$$\frac{\partial L}{\partial k} = 2 \times [y(w) - RNN(w)] \times (-1) \times \frac{\partial Adder^{+}(o_w)}{\partial k} \times \frac{\partial Adder(o_w)}{\partial o_w}$$

$$\frac{\partial L}{\partial k} = 2 \times [y(w) - RNN(w)] \times (-1) \times o_w \times k$$

Proposed Network Architecture

- Based on RNN, contains two parts: Recurrent part, and the adder

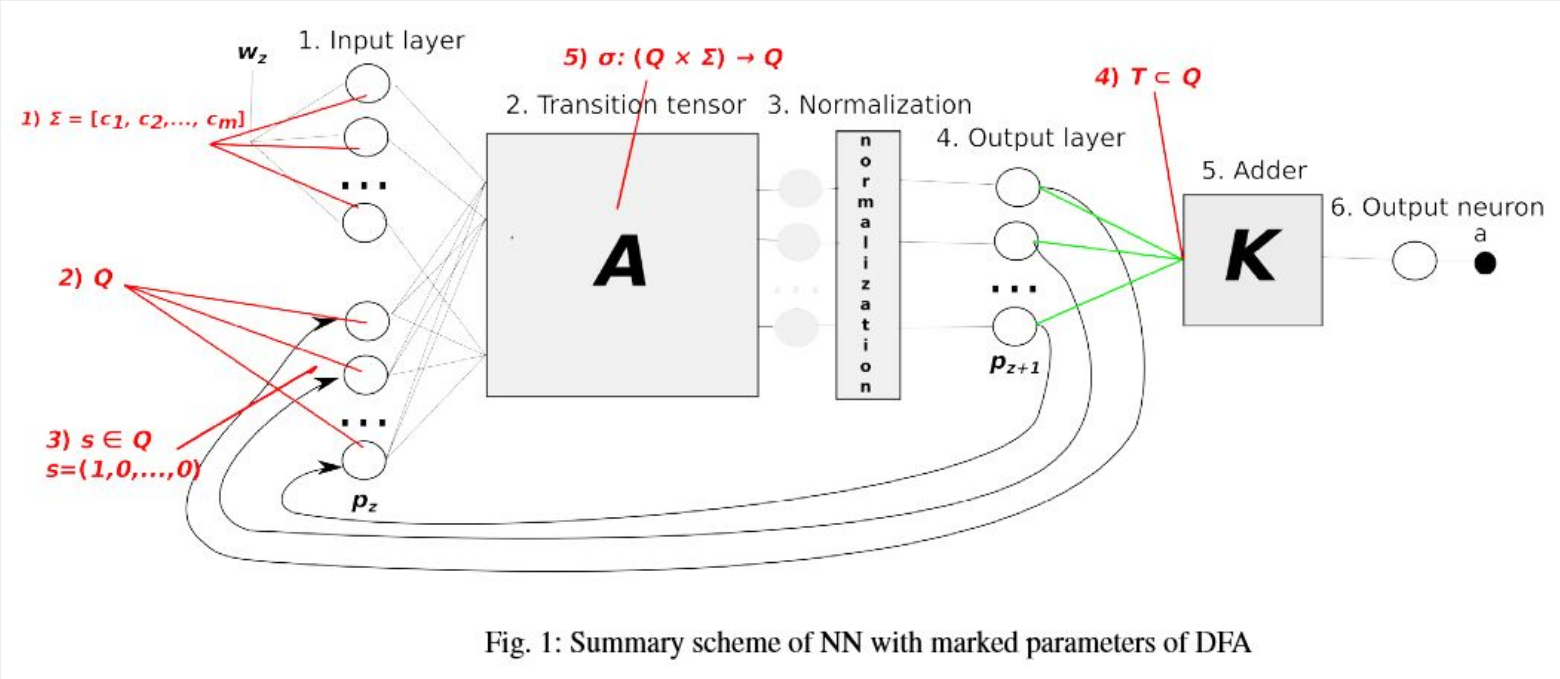


Fig. 1: Summary scheme of NN with marked parameters of DFA

Source: Paper

Backpropagation: Updating transition tensor *A*

- The loss function is *L*(y(w), *RNN*(w)) = (y(w) - *RNN*(w))^2
- While updating tensor weight *A*, we use:

$$A_{kji}(new) = A_{kji}(old) - \eta \times \frac{\partial L}{\partial A}$$

- We compute the derivative term using back propagation:

$$\frac{\partial L}{\partial A} = 2 \times [y(w) - RNN(w)] \times (-1) \times \frac{\partial RNN(w)}{\partial A}$$

- RNN(**w**) = **a** = *Adder*(**o**$_w$) = *Adder*(*Recurrent*(**o**$_{w-1}$, **w**$_w$))

  = *Adder*(*Recurrent(Recurrent*(**o**$_{w-2}$, **w**$_{w-1}$), **w**$_w$)

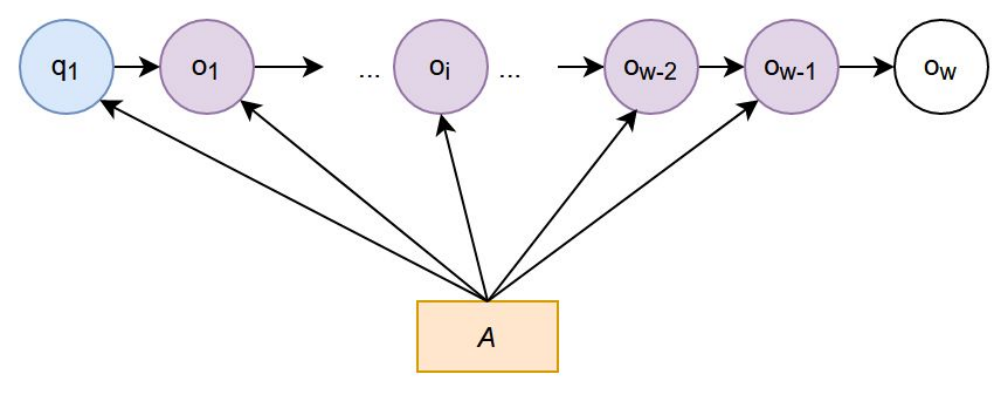
  = *Adder*(*Recurrent*(...*Recurrent*(**q**$_1$, **w**$_1$)...,**w**$_w$))

- Let e = y(w) - RNN(w) and k= adder derivative
- Also we know $\mathbf{o}_{i+1} = Recurrent(\mathbf{o}_i, \mathbf{w}_{i+1}) = Normalize(\boldsymbol{h}_i)$

$= Normalize(Match(\mathbf{o}_i, \mathbf{w}_{i+1}))$

$$
\begin{aligned}
\frac{\partial L}{\partial A} &= -2 \times e \times \frac{\partial RNN(w)}{\partial A} \\
&= -2 \times e \times \frac{\partial Adder(o_w)}{\partial A} \\
&= -2 \times e \times \frac{\partial Adder(o_w)}{\partial o_w} \times \frac{\partial o_w}{\partial A} \\
&= -2 \times e \times k \times \frac{\partial Recurrent(o_{w-1}, w_w)}{\partial A} \\
&= -2 \times e \times k \times \frac{\partial Normalize(h_w)}{\partial h_w} \times \frac{\partial h_w}{\partial A} \\
&= -2 \times e \times k \times \frac{\partial Nor(h_w)}{\partial h_w} \times \frac{\partial Match(o_{w-1}, w_w)}{\partial A}
\end{aligned}
$$

- Now, to be find derivative of *Match*($\mathbf{o_{w-1}}$, $\mathbf{w_w}$) with respect to *A,* we can't treat $\mathbf{o_{w-1}}$ as a constant as it is also dependent on *A.*



- For example, let's take a *Match*(o_3), and try to calculate its total derivative with respect to *A*. By chain rule, total derivative consists of two parts: explicit and implicit. In explicit part which is denoted by (+), we treat o_i's as constants, and in the implicit part, we find derivative wrt to o_i's.

- Let *Match* function be denoted by *M* and *Normalize* function be denoted by *N*

$$
\begin{aligned}
\frac{\partial M(o_3)}{\partial A} &= \frac{\partial M^+(o_3)}{\partial A} + \frac{\partial M(o_3)}{\partial o_3}\frac{\partial o_3}{\partial A} \\
&= \frac{\partial M^+(o_3)}{\partial A} + \frac{\partial M(o_3)}{\partial o_3}\frac{\partial N(h_3)}{\partial h_3}\frac{\partial M(o_2)}{\partial A} \\
&= \frac{\partial M^+(o_3)}{\partial A} + \frac{\partial M(o_3)}{\partial o_3}\frac{\partial N(h_3)}{\partial h_3}\left[\frac{\partial M^+(o_2)}{\partial A} + \frac{\partial M(o_2)}{\partial o_2}\frac{\partial N(h_2)}{\partial h_2}\frac{\partial M(o_1)}{\partial A}\right] \\
&= \frac{\partial M^+(o_3)}{\partial A} + \frac{\partial M(o_3)}{\partial o_3}\frac{\partial N(h_3)}{\partial h_3}\frac{\partial M^+(o_2)}{\partial A} + \frac{\partial M(o_3)}{\partial o_3}\frac{\partial N(h_3)}{\partial h_3}\frac{\partial M(o_2)}{\partial o_2}\frac{\partial N(h_2)}{\partial h_2}\frac{\partial M(o_1)}{\partial A} \\
&= \frac{\partial M^+(o_3)}{\partial A} + \frac{\partial M(o_3)}{\partial o_3}\frac{\partial N(h_3)}{\partial h_3}\frac{\partial M^+(o_2)}{\partial A} + \frac{\partial M(o_3)}{\partial o_3}\frac{\partial N(h_3)}{\partial h_3}\frac{\partial M(o_2)}{\partial o_2}\frac{\partial N(h_2)}{\partial h_2} \\
&\qquad \left[\frac{\partial M^+(o_1)}{\partial A} + \frac{\partial M(o_1)}{\partial o_1}\frac{\partial N(h_1)}{\partial h_1}\frac{\partial M^+(q_1)}{\partial A}\right] \\
&= \frac{\partial M^+(o_3)}{\partial A} + \frac{\partial M(o_3)}{\partial o_3}\frac{\partial N(h_3)}{\partial h_3}\frac{\partial M^+(o_2)}{\partial A} + \frac{\partial M(o_3)}{\partial o_3}\frac{\partial N(h_3)}{\partial h_3}\frac{\partial M(o_2)}{\partial o_2}\frac{\partial N(h_2)}{\partial h_2}\frac{\partial M^+(o_1)}{\partial A} \\
&\qquad + \frac{\partial M(o_3)}{\partial o_3}\frac{\partial N(h_3)}{\partial h_3}\frac{\partial M(o_2)}{\partial o_2}\frac{\partial N(h_2)}{\partial h_2}\frac{\partial M(o_1)}{\partial o_1}\frac{\partial N(h_1)}{\partial h_1}\frac{\partial M^+(o_0)}{\partial A} \\
&= \frac{\partial M^+(o_3)}{\partial A} + \sum_{i=0}^{2}\left(\prod_{j=i+1}^{3}\frac{\partial M(o_j)}{\partial o_j}\frac{\partial N(h_j)}{\partial h_j}\right)\frac{\partial M^+(o_i)}{\partial A}
\end{aligned}
$$

- Let d = -2 x e x k, then derivative of L wrt *A* is

$$
\begin{aligned}
\frac{\partial L}{\partial A} &= -2 \times e \times \frac{\partial RNN(w)}{\partial A} \\
&= -2 \times e \times \frac{\partial Adder(o_w)}{\partial A} \\
&= -2 \times e \times \frac{\partial Adder(o_w)}{\partial o_w} \times \frac{\partial o_w}{\partial A} \\
&= -2 \times e \times k \times \frac{\partial Recurrent(o_{w-1}, w_w)}{\partial A} \\
&= -2 \times e \times k \times \frac{\partial Normalize(h_w)}{\partial h_w} \times \frac{\partial h_w}{\partial A} \\
&= d \times \frac{\partial Normalize(h_w)}{\partial h_w} \times \frac{\partial Match(o_{w-1})}{\partial A} \\
&= d \times \frac{\partial N(h_w)}{\partial h_w} \times \frac{\partial M^+(o_{w-1})}{\partial A} + \sum_{i=0}^{w-2} \left(\prod_{j=i+1}^{w-1} \frac{\partial M(o_j)}{\partial o_j} \frac{\partial N(h_j)}{\partial h_j} \right) \frac{\partial M^+(o_i)}{\partial A}
\end{aligned}
$$

- This algorithm is called backpropagation through time (BPTT) as we are backpropagation through all time steps.

Problem with BPTT:

- Obviously, we can compute the full sum in the derivative but it might increase the gradient (dL/dA) enormously such that it blows up.
- Also, it means a subtle changes in the initial conditions can potentially affect the outcome significantly. This might lead to disproportionate changes in the outcome and may not give us the desired model we initially wanted to estimate.
- Hence, the full sum is never calculated in practice. There are several ways to deal with this problem and one of the ways is to truncate the product part of the sum which the paper does.
- So, for each term in the summation, instead of calculating the whole product part, we just calculate the first term of the product.

$$\frac{\partial L}{\partial A} = d \times \frac{\partial N(h_w)}{\partial h_w} \times \frac{\partial M^+(o_{w-1})}{\partial A} + \sum_{i=0}^{w-2} \left(\prod_{j=i+1}^{w-1} \frac{\partial M(o_j)}{\partial o_j} \frac{\partial N(h_j)}{\partial h_j} \right) \frac{\partial M^+(o_i)}{\partial A}$$

$$\approx d \times \frac{\partial N(h_w)}{\partial h_w} \times \frac{\partial M^+(o_{w-1})}{\partial A} + \sum_{i=0}^{w-2} \frac{\partial M(o_{i+1})}{\partial o_{i+1}} \frac{\partial N(h_{i+1})}{\partial h_{i+1}} \frac{\partial M^+(o_i)}{\partial A}$$

Combining gradient descent and backpropagation

- Training Algorithm

While error > 0.01:

For each example:

Do a forward pass and obtain $o_1, o_2 \ldots o_w$

Calculate the derivative of loss with respect to Adder:

$$\frac{\partial L}{\partial k} = 2 \times [y(w) - RNN(w)] \times (-1) \times \frac{\partial Adder(o_w)}{\partial k}$$

Calculate the derivative of loss with respect to Tensor:

$$\frac{\partial L}{\partial A} = 2 \times [y(w) - RNN(w)] \times (-1) \times \frac{\partial RNN(w)}{\partial A}$$

Update both Adder and Tensor parameters:

$$k_i(new) = k_i(old) - \eta \times \frac{\partial L}{\partial k}$$

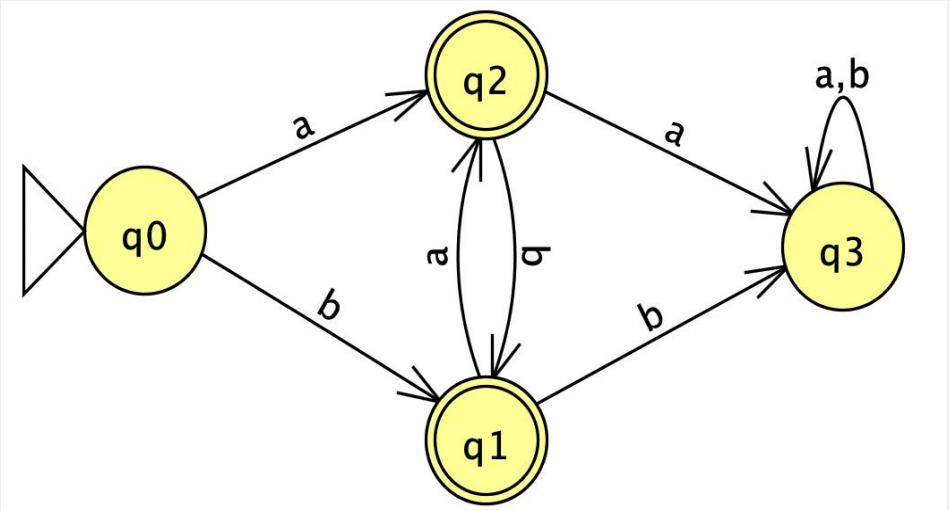
$$A_{kji}(new) = A_{kji}(old) - \eta \times \frac{\partial L}{\partial A}$$

*where η is the learning rate

Testing: language with |C| < 4 and |Q| < 6

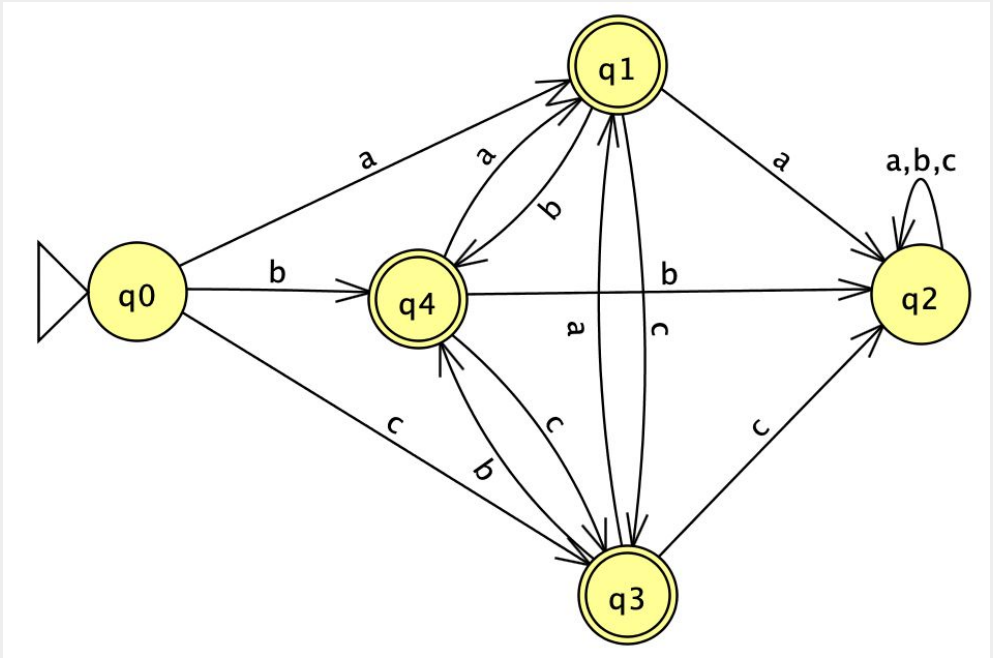
- Test Language L_1 = { w ∈ C = {a , b} | w has no same neighboring alphabets}
- RNN was successful in deriving a DFA for L_1

```
Epoch #1
Average error: 0.20892684414131055
Epoch #2
Average error: 7.90118595709727e-11
0--a-->2
0--b-->1
1--a-->2
1--b-->3
2--a-->3
2--b-->1
3--a-->3
3--b-->3
1 is terminal
2 is terminal
```



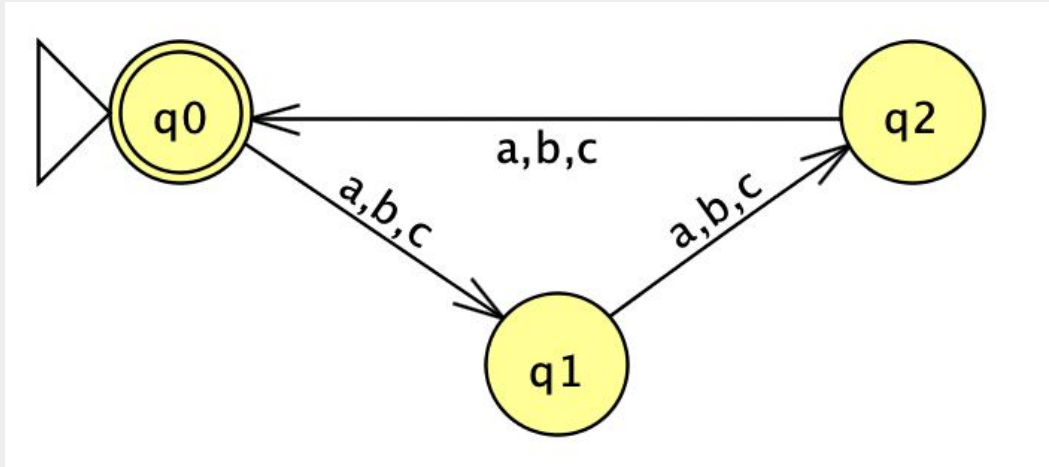
- Test Language L_2 = { w ∈ C = {a , b, c} | w has no same neighboring alphabets}
- RNN was successful in deriving a DFA for L_2

```
Epoch #4
Average error: 4.073875401090897e-10
0--a-->1
0--b-->4
0--c-->3
1--a-->2
1--b-->4
1--c-->3
2--a-->2
2--b-->2
2--c-->2
3--a-->1
3--b-->4
3--c-->2
4--a-->1
4--b-->2
4--c-->3
1 is terminal
3 is terminal
4 is terminal
```



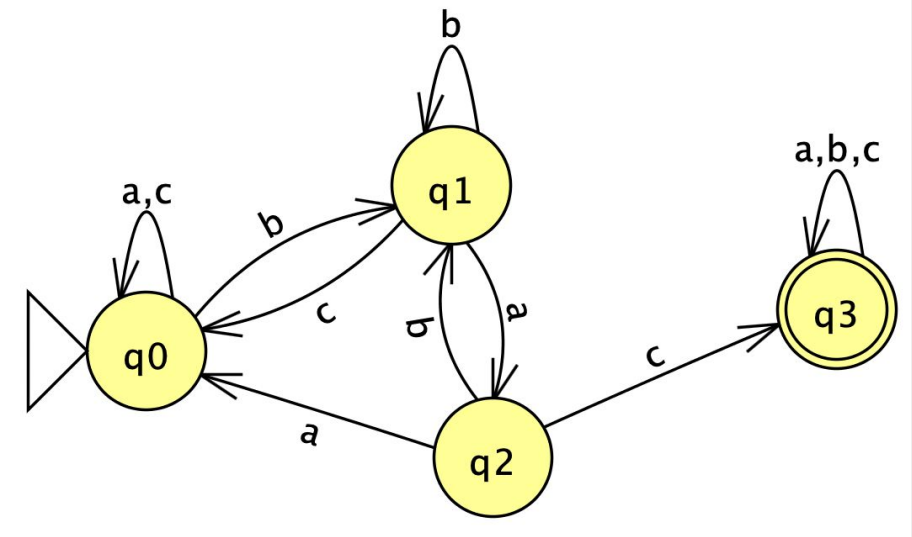
- Test Language L_3 = { w ∈ C = {a , b, c} | w has a length multiple of 3}
- RNN was successful in deriving a DFA for L_3

```
Epoch #3
Average error: 0.00084821672205610 16
0--a-->1
0--b-->1
0--c-->1
1--a-->2
1--b-->2
1--c-->2
2--a-->0
2--b-->0
2--c-->0
0 is terminal
```



- Test Language L_4 = { w ∈ C = {a , b, c} | w has ‘bac’ as a substring }
- RNN was successful in deriving a DFA for L_4

```
Epoch #2
Average error: 0.0030029774461518693
0--a-->0
0--b-->1
0--c-->0
1--a-->2
1--b-->1
1--c-->0
2--a-->0
2--b-->1
2--c-->3
3--a-->3
3--b-->3
3--c-->3
3 is terminal
```



- Test Language L_5 = { w ∈ C = {a , b, c} | w is given by the DFA M}
- RNN was successful in deriving a DFA *M'* for L_5

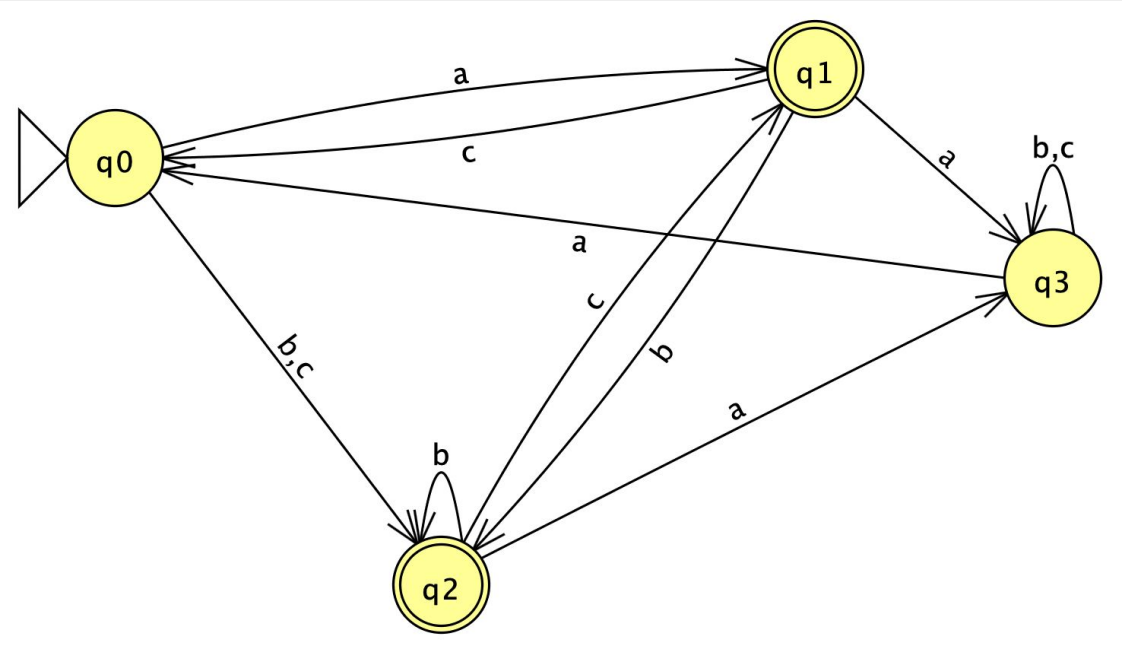


Fig 2. : DFA *M'*

```
Epoch #6
Average error: 1.1140296499681 25e-06
0--a-->3
0--b-->1
0--c-->1
1--a-->2
1--b-->1
1--c-->3
2--a-->0
2--b-->2
2--c-->2
3--a-->2
3--b-->1
3--c-->0
1 is terminal
3 is terminal
```

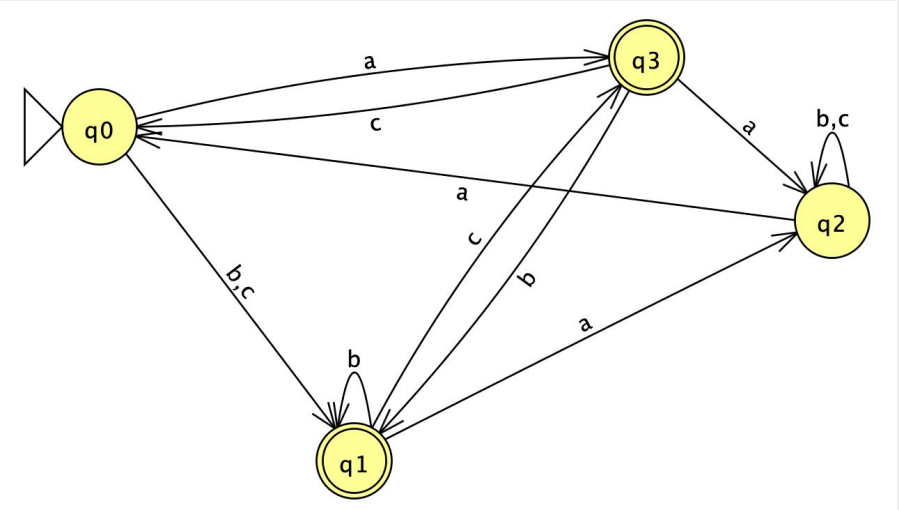


Fig 2. : DFA *M'*

- Test Language L_9 = { w ∈ C = {a, b} | w is of even length}
- RNN was successful in deriving a DFA *M* such that L(*M*) = L_9

```
Epoch #1
Average error: 0.00014382920040331267
0--a-->1
0--b-->1
1--a-->0
1--b-->0
0 is terminal
```

```
[[[0.          1.        ]
  [0.          0.98466574]]

 [[0.99998862 0.        ]
  [0.92794648 0.        ]]]
```

Fig 2. : Tensor *A* for DFA *M*

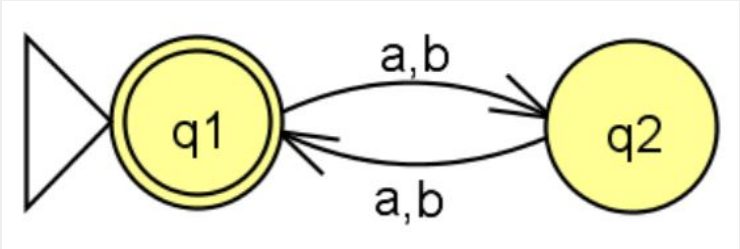


Fig 2. : DFA for L_9

```
[1.          0.01450799]
```

Fig 2. : Adder *K* for DFA *M*

Testing: language with |C| ≥ 4 or |Q| ≥ 6

1. Test language 6 i.e L_6 = { w ∈ C = {a, b} | w is of length 2 or 7}
   - State size greater than 5
2. Test language 7 i.e L_7 = { w ∈ C = {a, b, c} | w contains substring ‘abacaba’}
   - State size greater than 5
3. Test language 8 i.e L_8 = { w ∈ C = {a, b, c, d, e} | w is sorted lexicographically}
   - Alphabet size is greater than 3

➔ For these languages, the loss value < 0.01 could never be obtained and hence the RNN failed to provide a DFA.

Problem of Vanishing Gradient Descent:

- Initially, for languages with $|C| \leq 4$ and $|Q| \leq 6$, the proposed RNN suffered from the vanishing gradient descent problem.
- During backpropagation for an example, the absolute derivative value decreased at each step leading to near zero gradient.

  For example:

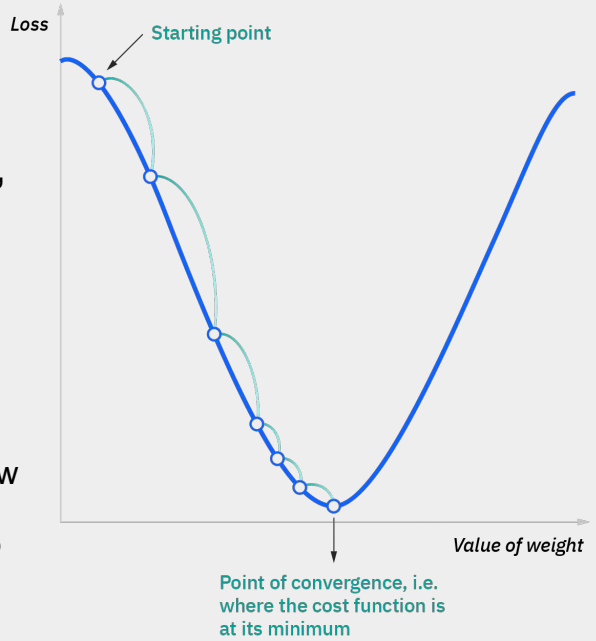
	o_1	o_2	…..	o_i	o_{i+1}	….	o_{w-2}	o_{w-1}	o_w
dL/dA =	0.0001	0.0005	…..	0.10	0.12	….	0.55	0.67	0.8

- This leads to characters at the end of a word having more influence on the training process that the characters at the beginning.

- When the gradient starts vanishing the change is loss wrt *A* is small for an example, hence a loss value < 0.01 can never reached to approximate the DFA

- The vanishing gradient problem is partially solved by fixing the gradient value at the first step and then for subsequent backwards step, the gradients are multiplied by a constant
- Constant at step i (k_i) = gradient at 1st step divided by gradient at step i

	...	o_i	o_{i+1}	...	o_{w-1}	o_w
dL/dA =	...	k_i x 0.10	k_{i+1} x 0.12	...	k_{w-1} x 0.67	0.8



Conclusion:

- The proposed RNN successfully generated the correct DFA where the no. of states < 6 and no. of characters < 4.

- The proposed RNN failed to generate DFA which required more states and/or more characters due to the vanishing/exploding gradient problem

- The paper claims that better models should be able to overcome this limitation of the RNN model. For example, principles from LTSM-network can be tested.

Thank You