Neural Network for Synthesizing DFAs

TOC Project - Presentation 01

Abhinav Nakarmi & Kuber Shahi

18th Apr’21

Project Description

- We are presenting the paper "*Neural Network for synthesizing Deterministic Finite Automata*" authored by "*Peter Grachev, Igor Lobanov, Ivan Smetanniko, Andrey filchenkov*"
- The paper proposes a new approach to build DFAs through neural network, specifically Recurrent Neural Network (RNNs).
- The proposed RNN trains on a labelled list of words/strings, which has fair amount of words belonging to a language, to approximate a DFA for the language.

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.

      For example: for language L = { w $\in$ Σ^* | Σ = { a, b} and w has no same neighboring characters}, then list = [(abab, 1), (abaa, 0),...]
    - If you have been given the list, then you don't need to know the language L but knowing it helps to confirm the outputted DFA.
2. **Output:**
    - DFA *M* such that L(*M*) = L

Implementation Demonstration

1. For regular language L_1 = { w ∈ Σ* | Σ = { a, b} and w has no same neighboring characters}, then list = [(abab, 1), (abaa, 0),...]

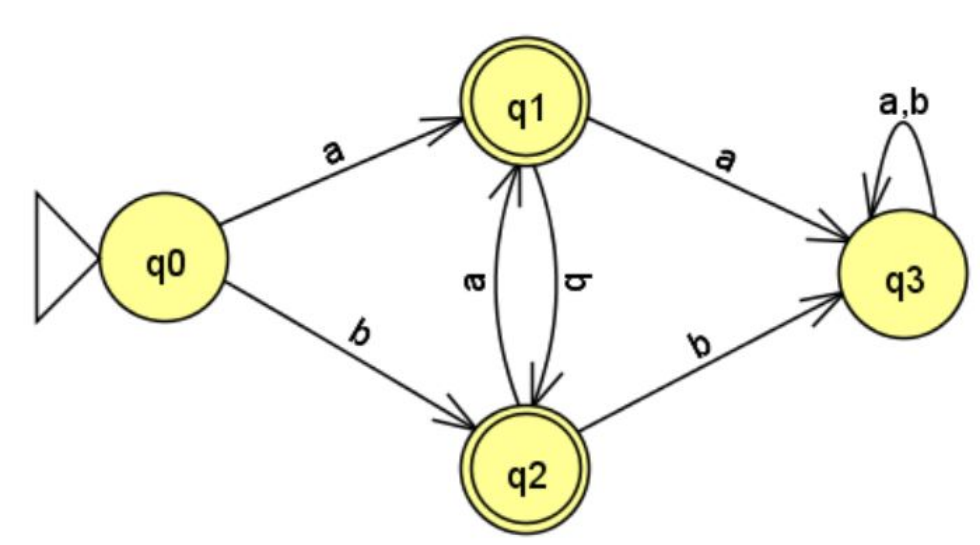


Fig 1.1: DFA *M* such that L(*M*) = L_1

Background

- The background knowledge required for the project are:
  - Understanding of Deterministic Finite Automata: we have avoided a detailed discussion of DFAs since we have already done this in class.
  - Understanding of Neural Networks (NNs): we have discussed the core ideas of what NNs are and what they do
  - Understanding of RNNs : we have discussed how RNNs extend the idea of NNs in solving problems that NNs fail to do.

DFA (Deterministic Finite Automata)

- DFA: an abstract mathematical model given by five tuple: $(Q, \Sigma, \delta, q_0, F)$ where Q is set of states, Σ is set of alphabet, δ is set of transition rules, q_0 is the start state, and F is the set of accepting states

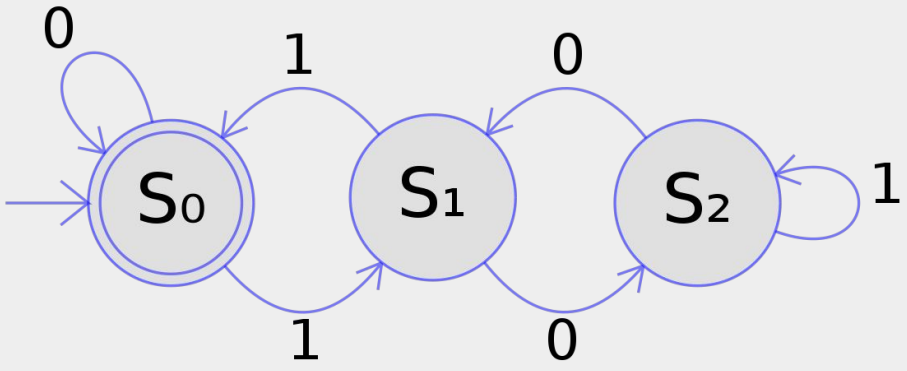


Fig 1.2: DFA *M* that accepts only binary numbers that are multiple of 3. (Source: Wikipedia)

Neural Networks (NNs)

Problem statement:

- Given a training data set comprising *N* observation of $\{x_n\}$, where n =1,...,N together, with their corresponding target values $\{y_n\}$, the **goal** is to predict the values of y for new values of x.
- To do this, we need to construct an appropriate function f(x) such that for new values of x, it predicts the corresponding values of y.

  $f(x_n; w_n) = y_n$, where x_i's are inputs, w_i's its coefficient, and y_i's its predicted outputs. Similarly, $x_i \in R^N$, $y_i \in R^k$ where N, k are natural number and f: $R^N \longrightarrow R^k$
- But what kind of function is f ? We really don't know, hence it is safe to assume it as a nonlinear function of input values x.

- **f(x; w)** as a nonlinear function of x can be written as:

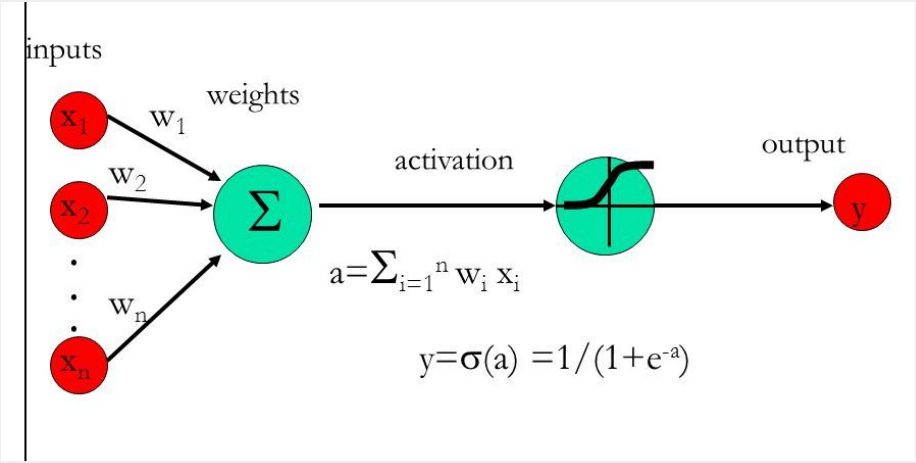
$$f(x, w) = \sum_{j=1}^{M} w_j \phi_j(x)$$

  where Φ(x) are fixed nonlinear functions of input values x.

- How might we generate such function for any given problem?
- Idea! : If we have enough (x_n, y_n) values, we might be able to approximate such a function f(x; w) with the help of a **Neural Network**.
- **Input:** *N* observations of (x_n, y_n). Each pair of (x_i , y_i) is called an example.
- **Process:** the NN trains on the *N* observations of (x_n, y_n) with the help of gradient descent and backpropagation algorithm
- **Output: f (x ; w)** where **f** is an approximation of **f(x; w)**.
- Then, apply **f** on new values of x to predict their corresponding y values.

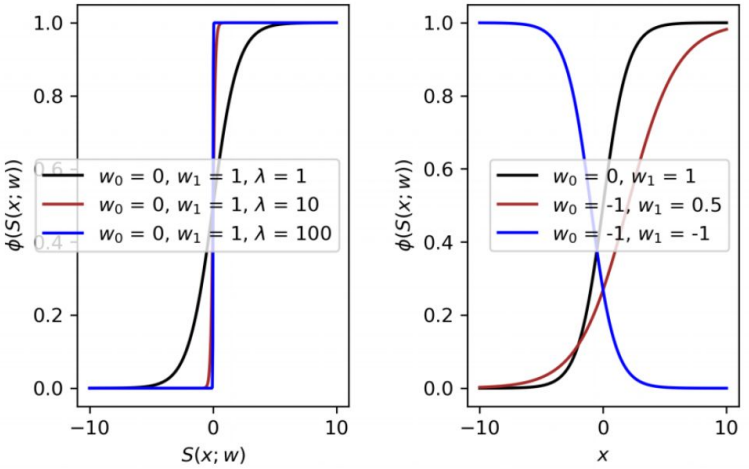
Building Blocks - Neural Network

A Perceptron/Neuron: takes several inputs like x_1, x_2,...., and produces a single output.



Weights: real numbers expressing importance of the respective input to the output

Activation Function: to introduce non-linearity into the output of a neuron.



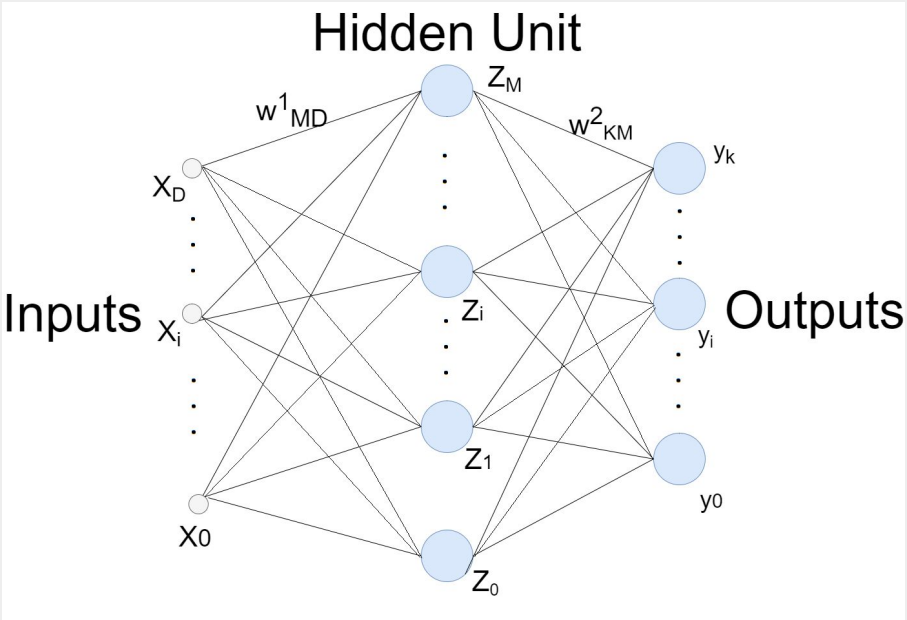
$$\hat{y}_i(x; w) = \phi\left(S_i(x; w)\right) = \frac{1}{1 + e^{-\lambda S_i(x;w)}}$$

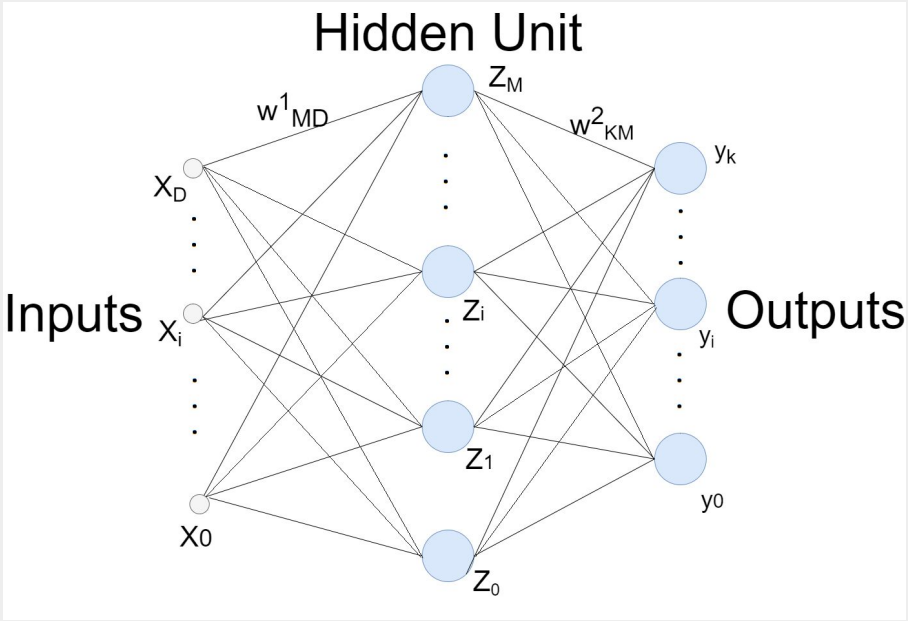
- Activation functions are typically nonlinear function which produces the neuron's output. Note: Above represents binary sigmoid function, where output is bounded between 0 and 1.

Output: Weighted sum i.e. $\Sigma_{j=1}\, w_j x_j$ for 0<j<n where n the total number of inputs over a activation function.

Neural Network

- An artificial neural network is an interconnected group of neurons.
- These neurons are connected by weights(w)
- Each neuron is a function of a sum of the inputs to the neuron activated by some activation function.





$$a_j = \sum_{n=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

- a_j represents weighted sum of a neuron in the hidden layer (Z)
- w_{ji} is the weight from i-th neuron of input layer to j-th neuron of hidden layer(Z) and w_{j0} is weight for bias unit (x_0)
- Then, we apply some nonlinear activation function *h* on a_j.

$$z_j = h(a_j)$$

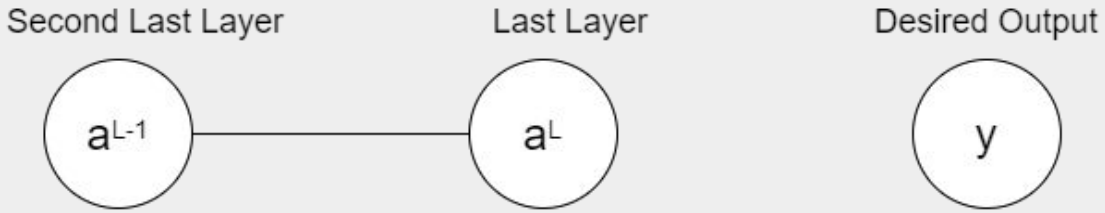
- Now, we combine these various nonlinear z_j to give the overall network function:

$$f_k(x, w) = \phi(\sum_{j=1}^{M} w_{kj}^{(2)} h(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}) + w_{k0}^{2})$$

where Φ is binary sigmoid function

Training the NN: Calculation magnitude of error (loss)

- Training the Neural Network
  - Learning = updating weights.

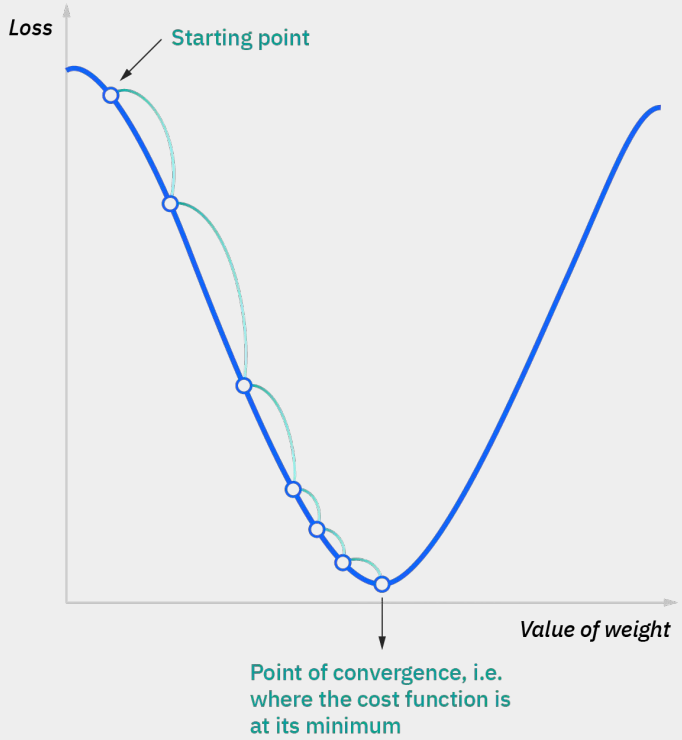


- Cost function: $C = (a^{L} - y)^2$
- The result of this cost function represents the magnitude of error for our neural network.
- Our goal now is to minimize this error. How?

Idea! **Gradient Descent** !

Gradient Descent: way to update weights

- Used to find local minimum of an differentiable cost function in terms of weight:
  - Start at any arbitrary point
  - We find derivative/slope
  - The derivative will inform the update to the weights!
  - Slope at start is likely to be steeper.
  - However, it will be less so as we reach the minimum point.



$$w_{ij}(new) = w_{ij}(old) - \eta \frac{\partial C}{\partial w_{ij}}$$

where η is the learning rate

Backpropagation: way to compute derivatives

Cost of an example: $C = (a^L - y)^2$

Cost derivative: $\frac{\partial C}{\partial a^{(L)}} = 2(a^L - y)$

Output for a layer a^L is given by: $a^{(L)} = \phi(z^{(L)})$

Output layer derivative: $\frac{\partial a^{(L)}}{\partial z^{(L)}} = \phi\prime(z^{(L)})$

Weighted sum z^L given by: $z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$

Weighted sum z^L derivative: $\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$

Combining all terms by chain rule!

$$\frac{\partial C}{\partial w^L} = \frac{\partial z^L}{\partial w^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial C}{\partial a^L}$$

Backpropagation Algorithm(single layer):

$$\frac{\partial C}{\partial w^L} = \frac{\partial z^L}{\partial w^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial C}{\partial a^L}$$

For each example:

Do a forward pass and obtain $y_1, y_2 \ldots$

For each i in number of output neurons:

For each j in number of input neurons:

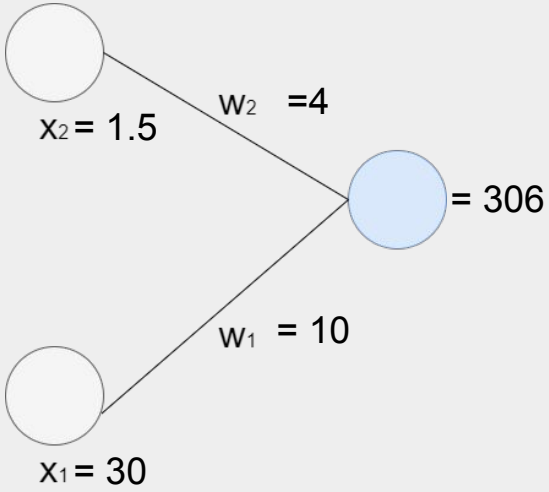
$$\frac{\partial C}{\partial w_{ij}} = a_j^{(L-1)} \cdot \phi' z_i^{(L)} \cdot (a_i^L - y_i)$$

$$w_{ij}(new) = w_{ij}(old) - \eta \frac{\partial C}{\partial w_{ij}}$$

*where η is the learning rate

- **Epoch** is a parameter that defines the number times that the **learning** algorithm will train through the entire training dataset.
- Therefore, while training a value of epoch is chosen such that the training is repeated some number of times which gives a cost value ≥ minimum cost.

Simple example of neural network



Inputs $x_i \in R$, and Output $y_i \in R$, we want to construct f(x;w) that predicts the price of a car given mileage and mass

Train data: N observation of (x_i , y_i)

For i-th example: x_1 (Mileage) = 30, x_2 (mass) = 1.5 (in thousands), w = [10 4], y_i = 300

So, let's update the weights

1) Forward pass:

$$a = x_2 w_2 + x_1 w_1 = 1.5 \times 4 + 30 \times 10 = 306$$

2) Cost function derivation:

$$\frac{\partial C}{\partial w_{ij}} = a_j^{(L-1)} \cdot (a_i^L - y_i)$$

Cost derivative(w_1) = 30 x (306 - 300) = 180

New w_1 = old w_1 - η x cost derivative

= 10 - 0.01 x (180) = 8.2

Cost derivative (w_2) = 1.5 x (306 - 300) = 9

New w_2 = old w_2 - η x cost derivative

= 4 - 0.01 x (9) = 3.91

In conclusion:

- Neural network is essentially a collection of units that jointly implement a highly complex (non-liner) function that maps a given input to an output. This network learns this functional mapping through the training on data.
- We struggle to explain why a particular prediction is done because the correlations within the data are not based on well-understood principles, rules and criteria. (J.C Bjerring)
- As a result, Neural Networks are regarded as Black-Box.

Recurrent Neural Networks (RNNs)

- So far, we saw what feed-forward Neural Networks are capable of doing. But like any model, it has limitations as well.
- Primarily, from viewpoint of our problem, it can’t handle sequences in the input data. We want a model that can process each character in a word i.e w = abab.

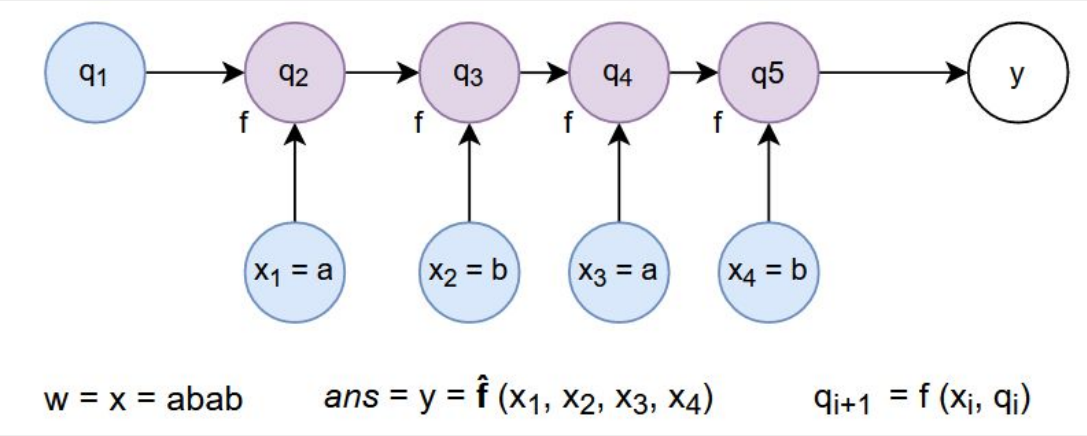


Fig: Overview of our problem statement

- In some sense, we want a model that can handle the dependence on prior inputs within the sequence to make the next prediction. Recurrent Neural Network does exactly this: it accounts for the dependence between the sequence of the input.
- It extends the idea of neural network by feeding the outputs back to the input to make the next prediction.

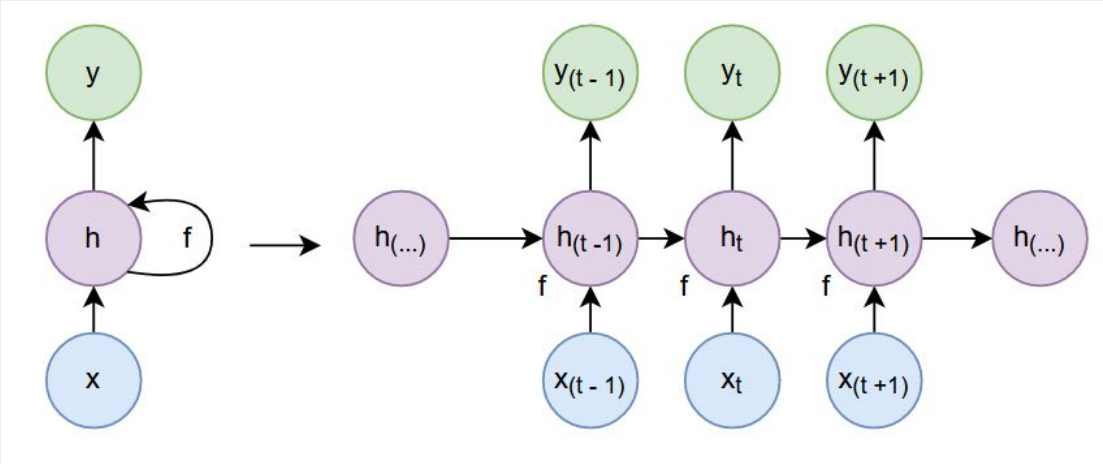


Fig : Top-level visualization of RNN

- The function that the RNN approximates is:

$$y_t = f(x_t, h_{t-1}; w_t)$$

- y_t is the output and x_t is the input at the step *t*. The output at the step *t* (y_t) is dependent on the inputs that the model has seen up to the step *t*.
- The task at each step remains the same (to predict the next y_t) but the input at each step changes. At each step *t*, same function *f* is being executed, where *f* is a polynomial function we want to approximate.
- RNNs are widely used in Natural Language Processing like Auto completion of words, parts of speech tagging, caption generation, etc as they have sequence data. Modelling on sequence data is called Sequence Modelling/Learning.

Example of Sequence Modelling Problem

- Part of Speech Tagging
  - Given a sentence ***s***, tag the words/string of ***s*** with its appropriate parts of speech

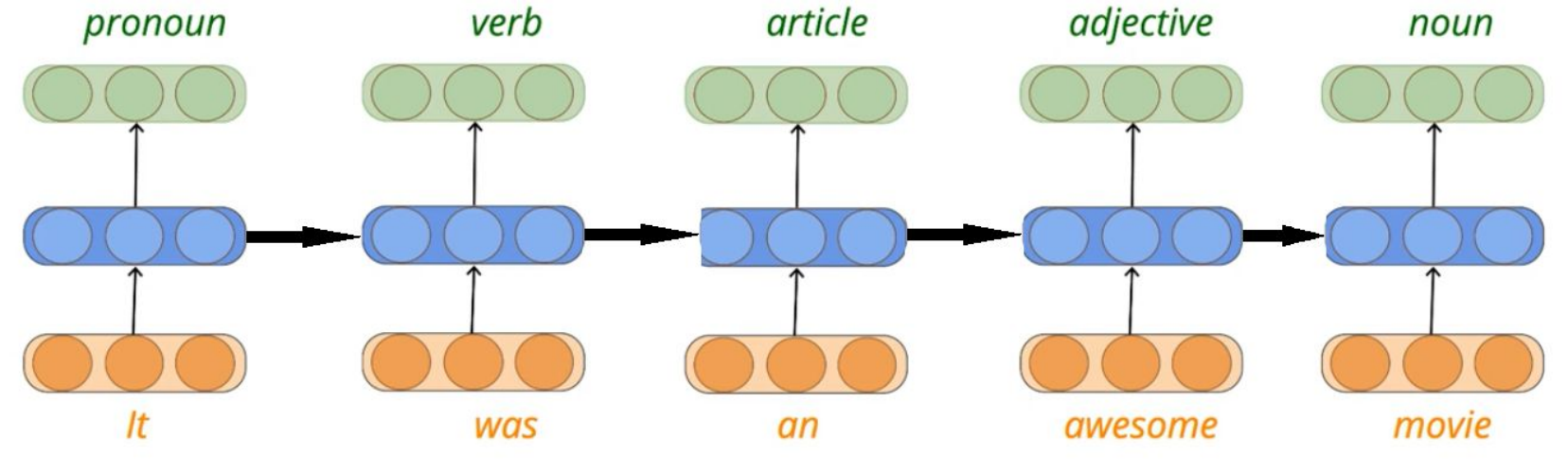


Fig: Part of Speech Tagging Problem (source: towards data science)

Learning in RNNs

- Only the architecture of the RNN is different from NN but the core ideas of approximating an unknown function through learning on labelled data remains the same.
- Hence, we still use stochastic gradient descent for learning i.e updating weights with backpropagation as the method of computing gradients.
- But now, since RNN involves step-by-step feed-forwarding to predict the next output, loss and backpropagation is done at each step.
- The loss function is calculated at each step *t*, Then, *L* of all *T* steps is defined as:

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T} \mathcal{L}(\hat{y}_t, y_t)$$

- Where L is any loss function, y_t is true output, and $\hat{y}_t$ is the predicted output at step *t.*

- Similarly, backpropagation is done at each step *t.* At step *N,* the derivative of loss *L* with respect to weight matrix *W* is expressed as sum of derivative at each step.

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial W}$$

Proposed Network Architecture

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

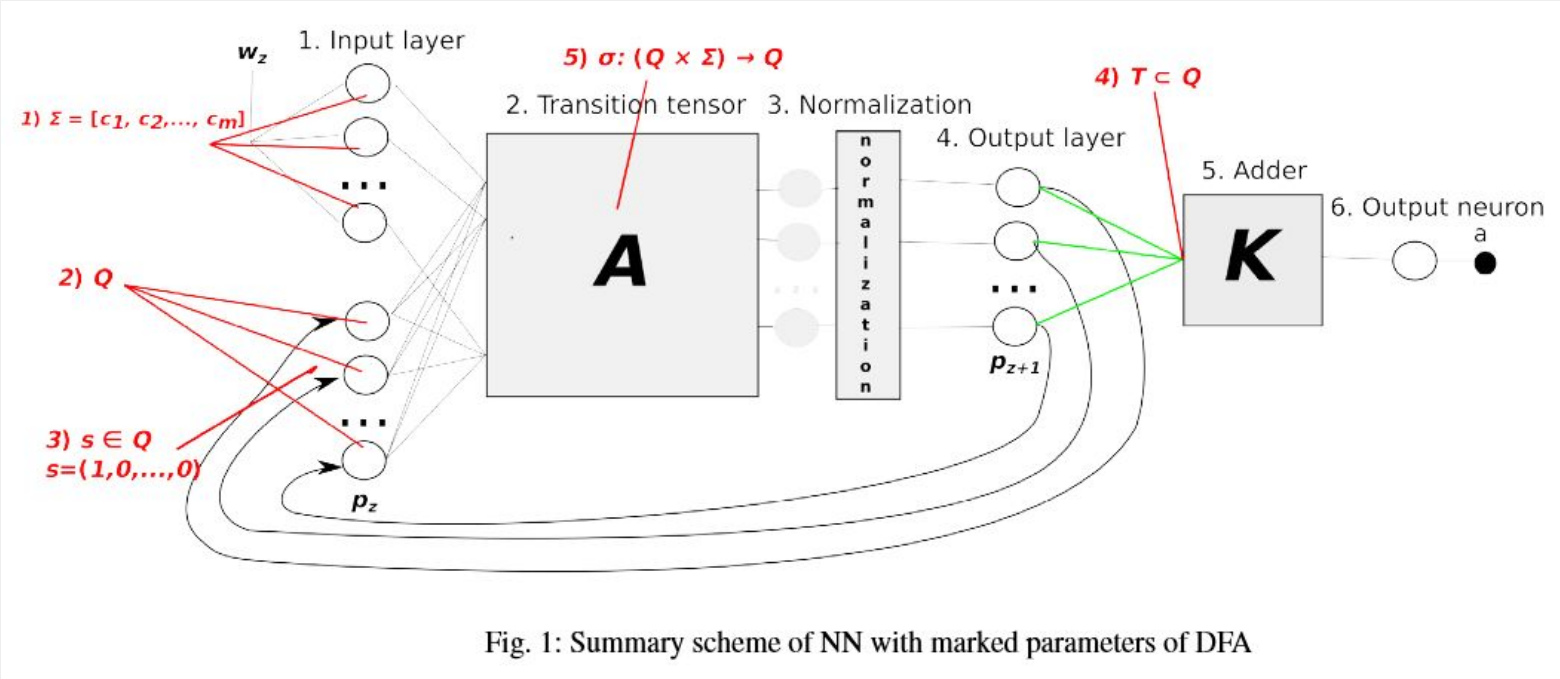


Fig. 1: Summary scheme of NN with marked parameters of DFA

Source: Paper

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 proposed RNN on the labelled data (list of tuples)
    - 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

Recurrent Part: Input Layer

- Consists $|C| + |Q|$ neurons where C is the alphabet size of the language and Q is the number of state in DFA (Note: the paper assumes the number of states for some language as an upper bound).
- First $|C|$ neurons determine the next input character of $\boldsymbol{w}$. Each character is treated as $|C|$-dimensional vector, say $\boldsymbol{c}$.
- For a character, components of vector $\boldsymbol{c}$ equals zero except for the one that corresponds to the index of the character in the alphabet set.
- For example, for A = { a, b, c }, a = [1 , 0 , 0], b = [0, 1 , 0], c = [0 , 0 , 1]
- $\boldsymbol{w}$ = aab is represented as [[1, 0 , 0]
  [1, 0, 0]
  [0, 1, 0]] dim: $|\boldsymbol{w}|$ x $|C|$

Input Layer

- Last |*Q*| neurons determine the current state. Each state is treated as |*Q*|-dimensional vector, say ***q*** .
- For a state, components of vector ***q*** equals zero except for one index that corresponds to its state number.
- For example, for |*Q*| = 5, q_1 = [1, 0, 0, 0, 0] is the start state, similarly

  q_2 = [0, 1, 0 , 0, 0] is the second state, and so on ...
- While training, the values in ***q*** apart from the first state are not purely discrete but continuous. Each index have values between 0-1 which can be interpreted as a probability distribution. For example,
  q_i = [0.035, 0.9, 0.025 , 0.025, 0.015] is q_2 with 0.9 probability.
- The input layer now represents two things of a DFA: *C*/Σ - set of alphabet, and *Q* - set of states.

Transition layer

- The output signals of the input layer neurons are fed into transition layer, and converted using tensor, say A_{ijk}, with dimensions $|Q|\times|C|\times|Q|$.
- As a result of applying A, a $|Q|$-dimensional vector h is obtained, k-th element of which is defined as:

$$h_k = \sum_{i=1}^{|Q|} \sum_{j=1}^{|C|} A_{ijk} \cdot q_i \cdot c_j$$

where q_i are the states in Q, and c_j are the characters in the input string w.

- At the end, **h**, a $|Q|$-dimensional vector is the next state after transition using the next character.
- Components of A_{ijk} are alterable parameters (weights) and hence needs to be trained

Understanding tensor *A*

- A_{kji} can be interpreted as the probability of existence of a transition **from i-th state to k-th state using j-th character** of the alphabet.
- See tensor *A* as a transition function, A takes an input character c_j and a state q_i, and performs a transition on them.
- *A* is a 3-dimensional tensor of dimension |*Q*| x |*C*| x |*Q*|. Hence, see A_{kji} *as* A[to][ch][from] i.e transition of form |*Q*| ← |*C*| x |*Q*|, 'ch' is the character, 'from' is the current state, and 'to' is the next state given by the transition

Algorithm to calculate h_k given *c, and q*

each k in (number of states):

each j in (number of alphabets):

each i in (number of states):

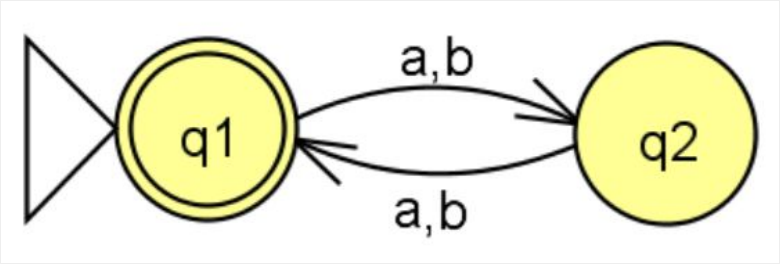
h[k] += tensor[k][i][j] * c[j] * q[i]

$$h_k = \sum_{j=1}^{|C|} \sum_{i=1}^{|Q|} A_{kji} \cdot c_j \cdot q_i$$

A transition tensor example:

- C = {a, b}, hence number of alphabets are $|C| = 2$. Let's assume a language

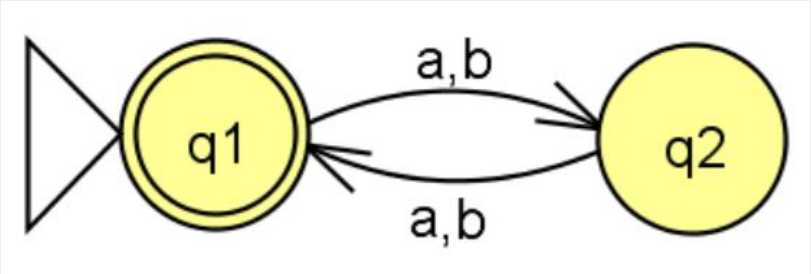
  L = { w $\in$ C^* | the length of w is even}



- Then, DFA M above is the DFA for L i.e L(M) = L
- This implies number of states are 2 i.e $|Q| = 2$
- Then, the dimension of transition tensor is A is 2 x 2 x 2 : (k x j x i)
- Let's create a transition tensor A that represents the transition rules for the DFA M

- One possible *A* can be:

$$\begin{bmatrix} \begin{bmatrix} 0.05_{(1a1)} & 0.95_{(1a2)} \\ 0.05_{(1b1)} & 0.95_{(1b2)} \end{bmatrix} \\ \begin{bmatrix} 0.95_{(2a1)} & 0.05_{(2a2)} \\ 0.95_{(2b1)} & 0.05_{(2b2)} \end{bmatrix} \end{bmatrix}$$



- Now, let's perform a transition from q_2 = [0 1] and upon seeing the next character *c* as 'b', then b = [0 1]
- Suppose h is the output state given by transition δ (q_2 , b) = h.

$$h_k = \sum_{j=1}^{|C|} \sum_{i=1}^{|Q|} A_{kji} \cdot c_j \cdot q_i$$

$$\left[\begin{bmatrix} 0.05_{(1a1)} & 0.95_{(1a2)} \\ 0.05_{(1b1)} & 0.95_{(1b2)} \end{bmatrix} \begin{bmatrix} 0.95_{(2a1)} & 0.05_{(2a2)} \\ 0.95_{(2b1)} & 0.05_{(2b2)} \end{bmatrix} \right]$$

c = [0 1]
q = [0 1]

- Then, $h_1 = A_{111} \times c_1 \times q_1 + A_{112} \times c_1 \times q_2 + A_{121} \times c_2 \times q_1 + A_{122} \times c_2 \times q_2$
  = 0.05 x 0 x 0 + 0.95 x 0 x 1 + 0.05 x 1 x 0 + **0.95 x 1 x 1**
  = 0.95

  Again, $h_2 = A_{211} \times c_1 \times q_1 + A_{212} \times c_1 \times q_2 + A_{221} \times c_2 \times q_1 + A_{222} \times c_2 \times q_2$
  = 0.95 x 0 x 0 + 0.05 x 0 x 1 + 0.95 x 1 x 0 + **0.05 x 1 x 1**
  = 0.05
- Hence, h = [0.95 0.05] = q_1 with a probability of 0.95

- We assumed q_2 = [0 1] but in reality, except for the start state q_1 = [1 0] which we manually put, all the other states won't have discrete values but continuous between range [0-1].
- Hence, the sum of components of vector h (we saw before) could exceed 1.
- In order to interpret the output h again as a discrete probability distribution, we need to normalize it.
- Now, the transition tensor *A* represents the transition function of a DFA:
  $A = \delta : |Q| \times |C/\Sigma| \longrightarrow |Q|$

Normalization and the output layer

- The output signals of the transition layer ($\mathbf{h}_k$) which is the next state after performing transition is fed into normalization layer.
- In this layer, $\mathbf{h}_k$ is normalized giving output vector $\mathbf{o}_i$ which is calculated as:

$$\mathbf{o}_i = \frac{\mathbf{h}_i}{\sum_k \mathbf{h}_k}.$$

- Now, the vector **o** can be interpreted as discrete probability distribution
- Value $\mathbf{o}_i$ can be interpreted as the probability that the transition performed using the next input character resulted in *i*-th state.
- This vector **o** is then fed back to the input layer as the current state.

A normalization example:

- For example if h = [0.975 0.075], then the normalized output vector ***o*** would be ***o*** = [0.93 0.07]

Recurrent Part

- So far we discussed Input, Transition, Normalization and the output layer which together forms the repeating part of the proposed network.
- Performing a transition can be defined as a function *Match:* $R^{|C| + |Q|} \longrightarrow R^{|Q|}$ which takes the current state and the next input character, and outputs the next state. This is analogous to the transition (δ) of a DFA.
- Similarly, normalizing the next state can also be seen as a function *Normalize:* $R^{|Q|} \longrightarrow R^{|Q|}$
- Putting them together we have the recurrent part

$$Recurrent(\mathbf{x}) = Normalize(Match(\mathbf{x}))$$

Recurrent Part

- For a word **w,** which has |**w**| characters, the recurrent part of the proposed network runs |**w**|, at each iteration taking the current state and the next input character to output the next state.
- The recurrent part starts with the first character of ***w*** and the start state(q_1) and repeats |***w***| until it consumes every character of word reaching the end of the string.

  If $\mathbf{o}_w$ is the output state of the output layer after consuming the last character (*w*-th) character, then it can written as:

  $$\mathbf{o}_w = Recurrent(\mathbf{o}_{w-1}, \mathbf{w}_w) = \ldots = Recurrent(...Recurrent(\mathbf{q}_1, \mathbf{w}_1)...,\mathbf{w}_w)$$

  where $\mathbf{o}_i$ is the output vector of the recurrent part after consuming the *i*-th character of **w,** and $\mathbf{q}_1$ is the start state.

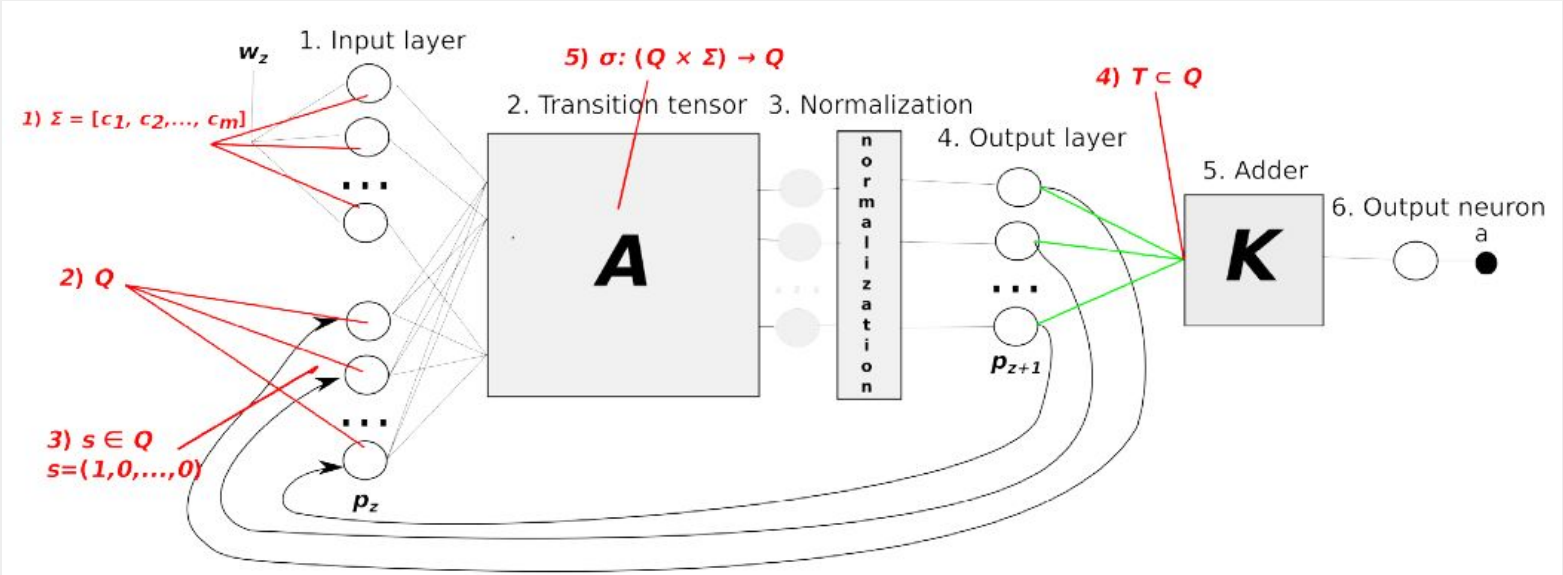


Fig. 1: Summary scheme of NN with marked parameters of DFA

Source: Paper

Adder Part

- Once the recurrent part has run |**w**| of times we get $\mathbf{o}_w$, and now we have to determine whether $\mathbf{o}_w$ is an accepting state or not. This is where adder function comes in.
- $\mathbf{o}_w$ is fed into the adder part which outputs a single scalar value **a** in the range [0,..,1].
- The value **a** is calculated through a = *Adder*(**o**) = < k, o> where **k** is the |*Q*|-dimensional parametric vector, each component of which is a scalar value in range [0-1].
- Value $\mathbf{k}_i$ can be interpreted as the probability of *i*-th state being an accepting state. For example: k = [0.05 0.95 0.95] which means that the second state and the third state are accepting states with 0.95 probability.

Adder Part

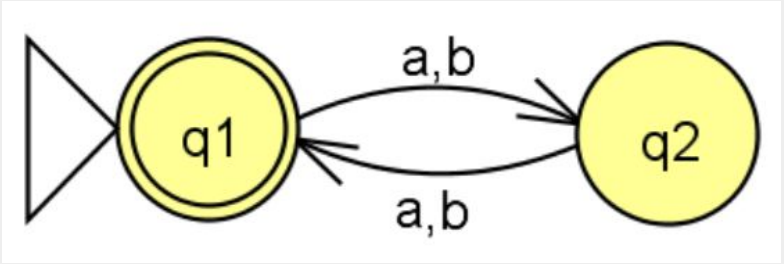
- The final output neuron value **a** in the range [0-1] can be interpreted as the probability that the input word belongs to the predetermines language.
- Now, the adder can be seen as a function *Adder:* $R^{|Q|} \longrightarrow R$
- The adder vector **k** now represents the set F of a set: **k** has information about which states are the accepting states.
- **k** is an alterable parameter of the network and hence needs to be trained.

An adder vector example:

- Let's use the same example we used for explaining transition tensor.
- The earlier example was:

  C = {a, b} and $|C| = 2$

  L = { w $\in$ C^* | the length of w is even}



- Then, DFA *M* above is the DFA for L i.e L(M) = L
- Number of states is 2 i.e $|Q| = 2$
- Then, the dimension of adder tensor is ***k*** is 2 x 1
- Let's create an adder tensor ***k*** that represents the information about accepting states in DFA *M*

- Since q_1 is the only accepting state in *M*, one possible vector ***k*** is

  k = [0.975 0.05]

- Then, suppose w = aabb which is even length and should end up in an accepting state i.e q_1. For that, the last transition needs to be performed on ‘b’ while the automaton is in state q_2.
- Earlier we performed the same transition in transition tensor *A* example i.e $\delta (q_2 , b) = \boldsymbol{o}_w$. The output $\boldsymbol{o}_w$ is the final state reached after consuming all the |w| characters of w.
- Earlier we had, $\boldsymbol{o}_w$ = [0.95 0.05]. Let’s apply tensor ***k*** on it to see if the w ended on an accepting state or not.
- $Adder(\boldsymbol{o}_w) = < k, \boldsymbol{o}_w > = 0.975 \times 0.95 + 0.05 \times 0.05 = 0.92$
- So, the final output is 0.92 which means the probability of $w \in L$ is 0.92 . As the probability is high we can say $w \in L$.

- Similarly, suppose $\boldsymbol{o}_w$ = [0.1 0.9] is the final state given by the proposed RNN when running on some string w .
- *Adder*($\boldsymbol{o}_w$) = < *k*, $\boldsymbol{o}_w$ > = 0.975 x 0.1 + 0.05 x 0.9 = 0.14
- So, the final output is 0.14 which means the probability of w $\in$ L is 0.92 . As the probability is low, we can say w $\notin$ L.

Combining Recurrent and Adder

- Now, the recurrent and the adder can be combined together to form the proposed RNN.
- Then, when we run the proposed RNN on a word **w,** the computation done by the RNN can be seen as:

RNN(**w**) = *Adder*($\mathbf{o}_w$)

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

= …

= *Adder*(*Recurrent*(...*Recurrent*($\mathbf{q}_1$, $\mathbf{w}_1$)...,$\mathbf{w}_w$))

= **a**

Training the Recurrent Neural Network

- So far we saw how the proposed RNN tries the mimic a DFA, mainly two components of a DFA, the transition function (δ) and the set of accepting states (F)
- To approximate the δ and F of a DFA *M* for the language L, we will train the proposed RNN on the labelled data (list of tuples) such that tensors *A* and k mimics the δ and F.
- To do so, we will use stochastic gradient descent with backpropagation to minimize the loss.
- The training will be stopped once an error(loss) lesser than an specified error is achieved. (want error of the proposed RNN to be < 0.01)
- We will discuss in-detail about the training and testing of the proposed RNN in the next presentation.

Thank You