# Recurrent Neural Networks - The way I understand it

Recurrent neural networks (RNN) is a variant of Deep Neural Networks (DNN), designed to help build models that take into account sequential relationship between the data points in the training data. Accounting for the sequential relation helps the model predict what is next based on what it already knows. Many authors use the word 'time' dimension for the sequential relation and I am going to avoid that as it misleads beginners to believe that to use this algorithm one needs to have time dimension in the training data. Which is not correct. One may or may not have a time dimension for a sequentially related data points. For e.g in Natural Language Processing (NLP), the word that appears next is dependent on the words used till now in a sentence (assuming the expression is syntactically and semantically correct). In this case each word is a data point that is sequentially related to preceding words if any and the next.

All models that we build represent processes in the real world. Process that may be tangible or intangible. Modelling the behaviour of a car in terms of mpg as a function of characteristics of a car, is an example of tangible process. Modelling customer behaviour thru market basket analysis is an example of an intangible process. Whatever the process may be, it should be consistent in their behaviour and the consistency should reflect in the data points it generates. Consistency relates to predictability.

There are two categories of modelling approaches. The **functional approach** where the relation between target and input variables is expressed as a function. For e.g. in Linear Regression the expression y = mX+ c + e, the relation between y and X expressed in terms of m , c and e. Mathematically, y is a function of X given m,c,e. Here consistency is expressed in terms of the distribution characteristics of the input and output data points.

All processes can be defined as sequence of logical steps executed to transform the input X into output y. The logical steps can be individually looked upon as sub-processes that take intermediate input and transform them to inter mediate output except for the first sub process which takes X as input and the last sub-process that emits y as output. Here consistency is reflected in terms of the allowed sequence/s of steps. For e.g. to express my ideas, am using English statements here and for the statements to be meaningful, the words forming the statements have to be in certain sequence. The sequence allowed is dictated by the grammar of the language. Given that, one can understand a process in terms of the internal sequence/s of steps also known internal states.

Thus, we can model the processes as **state machines (SM)** where a process goes thru different internal states in some logical order to carry out the overall task of converting X (input/s) to y (output/s) for e.g. X could be a sequence of words in Kannada being translated to y which could be a sequence of words in Hindi. The internal states are so called as they are not known or visible outside the process. The result of a process being in a particular state can be emitted and be visible as an output of the process.

The state machine moves from one state to next on occurrence of a trigger. The trigger may be external to the process such as a switch (On / Off) or internal such as a timer.

To define a process state objectively (in a quantifiable way), we use state variables. Each state of a SM can be defined / represented by a particular combination of state variable values. To understand the concepts of states, state variables, state transitions and trigger let us discuss the very common SM that most of us would be familiar with i.e. a fully automatic washing machine (**FAWM**).

Pic 1

**Note:** in this case the states of the system are observable and so are the state variables.

The following grid represents the different possible states (shown as row indices) a fully automatic washing machine can be in when it is processing. Each state is represented by a set of state variables (shown as column headers). The states and state variables can be expressed in finer terms if one likes. Irrespective of the degree of fineness, the concepts remain the same.

STATE VARIABLES

| STATE | State / Variables | Power (0/1) | Program 0, m(1-6) | Program Start (0/1) | Timer Trigger (0 /1) | Water Pump (0/1) | Water Level (0-L) | Outlet (0/1) | Spin (0 / n) |
|---|---|---|---|---|---|---|---|---|---|
| | Off | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Idle | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Programmed | 1 | m | 0 | 0 | 0 | 0 | 0 | 0 |
| | Program Execute | | | | | | | | |
| | Soak | 1 | m | 1 | 1 | 1 | L | 0 | 0 |
| | Rinse | 1 | m | 1 | 1 | 0 | L | 0 | n |
| | Drain | 1 | m | 1 | 1 | 0 | 0 | 1 | 0 |
| | Dry | 1 | m | 1 | 1 | 0 | 0 | 0 | n |

Pic 2

In this example all the state variables are binary in nature like a toggle switch. This need not be the case. We can have state variables which are continuous in nature or have more than two possible values such as the speed control on the washing machine (not shown in the example).

**FAWM cycle**

Let us discuss the washing process in terms of the state and state variables.

1. Let the initial state of the FAWM be the "off" state. In that state the state variables such as "Power" (which represents whether power switch is on or off), "Program" (whether program is selected or not), "Program Start" (whether the program is started or not) etc. are all shown with value of 0.
2. When the FAWM is turned on, it transitions from the "off" state to the "idle" state. The trigger for this state transition is external in from of a switch being turned on.
3. The "idle" state is defined in terms of the same state variables but with a change in value of the "Power" state variable
4. Next we select the program of choice. Assume there is a choice of six different program settings. We can choose any one of the six programs. For this we press the program of choice. The FAWM moves from "Idle" state to "Programmed" state and the trigger for this state transition is again an external trigger in form of program selection
5. The state "Programmed" is expressed using the same state variables but this time the value of "Program" state variable changed to "m" where "m" can take any value from 1 to 6. **Note:** the act of loading the clothes into the FAWM does not change the state of the machine and hence not discussed.
6. Once the FAWM is in "Programmed" state and ready to go, we click the "Start" button that transitions the FAWM to running mode in which the first state is to soak the clothes "Soak". Once again, the state is defined terms of the same state variables with the value of "Time Trigger", "Water Pump" and "Water Level" being set to appropriate values. The states and state variables can be expressed in much more detailed way than what is used here. The idea is to explain the concept so restricted the definitions at a gross level
7. The "Timer Trigger", "Water Pump", and "Level indicator", are **internal triggers**. These are not activated by any external agent interacting with the FAWM. Many such triggers exist to transition the FAWM states internally. For this discussion, I have taken only these two.
8. Finally, in the "Dry" state, the FAWM comes to a halt after completing the state transitions from start till the end. The result of this is the dirty linen come out of the process as clean dry clothes

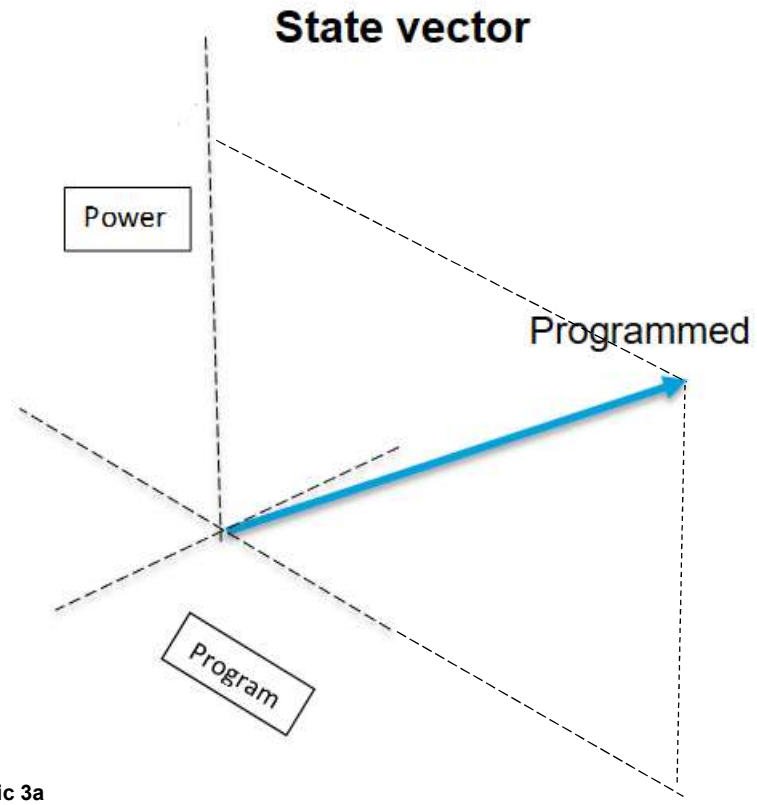
In this entire cycle, we represented the behaviour of the process using eight state variables. Thus, the feature space or the **state space** (as it is often called) is an eight dimensional space. Some authors use the word **Hilbert space**. We will refer to it as the state space. Thus every state in this example was represented as a vector with each

element of the vector being one state variable value. For e.g. Off state = [0,0,0,0,0,0,0,0], Idle state = [1,0,0,0,0,0,0,0].

The same information can also be represented geometrically representing these vectors as arrows in the state space emanating from the origin of the state space and the head of the arrow representing the point in space where the values of the attributes acquire the state variable values in the vector.

For e.g. the "programmed" state can be represented as shown below (Note: we can visualise only three dimensions though there are eight in this example).

Programmed state = [1, 1, 0, 0, 0, 0, 0, 0] (Row 3 in Pic2)

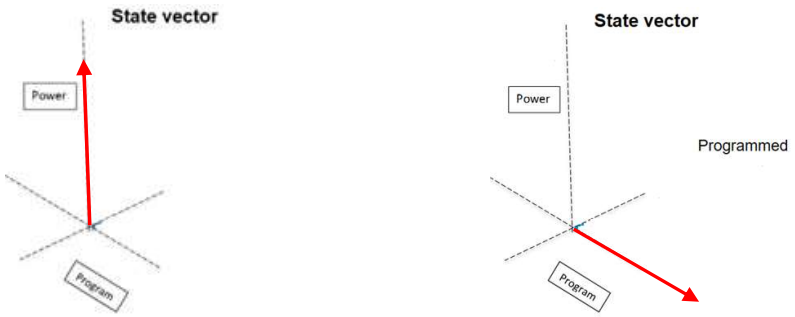



**Pic 3a**

The blue arrow visually represents the "Programmed" state vector in the state space (shown only three dimensions from eight) is also shown as [1, 1, 0, 0, 0, 0, 0, 0]. Each of the three dimensions in this state space represents a state variable. In this example the two dimensions represent two variables "Power", "Programmed". The third dimension is not used yet and could be "Program Start".

The length of the blue arrow vector is arbitrary in this example. When the state variables are continuous in nature instead of being toggle type, then the length of the vector will make sense.

The triggers (external and internal) can be represented by an arrow along the axis. For e.g. the trigger of switching on and selecting a program can be represented as a red arrow from origin on respective dimension.




[1, 0, 0, 0, 0, 0, 0, 0]  [0, 1, 0, 0, 0, 0, 0, 0]

**Pic 3b**

Elementwise addition of these two vectors i.e. [1, 0, 0, 0, 0, 0, 0, 0,] + [0, 1, 0, 0, 0, 0, 0, 0] results in [1, 1, 0, 0, 0, 0, 0, 0] which is the blue vector in the pic(3a) representing the FAWM in swtiched on and program selected.

This representation of triggers as vectors and vector addition leading to state vector transition thru simple elementwise addition is a very simplistic. When we deal with real world systems the triggers themsleves may be a combination state variable values instead of just one variable as in this example. Suppose we wish to train a robot to understand the FAWM as a state machine and operate it without any mistake.

**Training a Robot to run a WM**

We wish to train a robot to operate the FAWM. For that, imagine we have a log file generated by the FAWM (☺). The log file is a series of records with each record containing the current state, trigger and next state that the WM went thru all the times it was operated. For e.g.

| **Step** | **X1**<br>**Current State** | **X2**<br>**Trigger** | **y**<br>**Next State** |
|---|---|---|---|
| 1 | Off | Power | Idle |
| 2 | Idle | Program Select | Programmed |
| 3 | Programmed | Start | Soak |
| 4 | Soak | Level | Rinse |
| 5 | Rinse | Timer | Soak |
| 6 | Rinse | Timer | Drain |
| 7 | Drain | Timer | Dry |
| 8 | Dry | Timer | Stop |

**Pic 4**

The current state and the trigger are the inputs at a step and the dependent variable is the is the next state i.e. state at next time step. In the above table, there is a loop between soak and rinse states which is not shown.

Using this log file as training data, we can train a robot to run a FAWM (fire the external triggers and wait till the process gets over). This data has a sequential dependency (sequence of the steps). If the sequence is disturbed, training the robot will become impossible. When such sequential data is generated, another column or attribute is included that captures the proper sequence and helps maintain the sequence even if the order is disturbed by chance. I have shown the sequence column as the "Step" in the picture above.

When trained on a large set of such log files generated for the different programs at different times of operations, the robot is expected to learn the right sequence of operations. For e.g. while the FAWM is in rinse state, there is no effect of pressing the "select program" button i.e. there is no relation between the FAWM "Rinse" state and the external trigger "Select Program". If such an entry occurs in the training set, it will be noise probably generated by accident say when a child tries its luck pressing the buttons randomly.

The Robot should learn the correct valid state and trigger combinations and the sequence of state transitions. It should ignore the noise.

Thus the pattern learnt by the Robot could be represented as state equation -

***Next Sate = f (Current State, Trigger)***

i.e. next state is dependent on current state and trigger.

In this example, we know the various possible states the machine can take. However, in many situations, we may not have view of the various valid internal states in terms of what they are and how many, which state with which trigger is a valid combination etc. However, using **Implicit Function theorem** (out of scope for this post. Ref: https://en.wikipedia.org/wiki/Implicit_function_theorem#:~:text=In%20mathematics%2C%20more%20specifically%20in,functions%20of%20several%20real%20variables.&text=The%20implicit%20function%20theorem%20gives,there%20is%20such%20a%20function.), one can still model the state machine using the input (triggers) and output (predictions). The challenge is to guess how many state variable and internal states will be adequate to model the behaviour of the system.

**Introduction to Dynamic Systems**

All the models we built till now such as logistic, Naïve Bayes, CNN etc. represent process which on an input respond with an output. The output does not change as long as the input does not change. Such systems belong to feed-forward systems.

Some systems have the facility to feed the output back to the system along with the next input. In case of washing machine example, the "Next state" column captures the result of current state and input. This result becomes current state in the next step which along with relevant trigger leads to subsequent state.

Thus FAWM works in a sequence of steps with delay between the steps. Consequently, the information fed back after a step delay, modifies the overall input to the system leading to a new state. The system transits from state to state with time. Such systems are called **dynamic systems**.

**Dynamic Systems and State Space**

The state space is a continuum where every point in the space is a combination of certain values of the state variables. However, not all the points in the continuum represent valid states of the given system. Only certain combination of values of the state variables make a valid state. Such a point in the state space represents a valid state of the system and is also known as attractor. There is a one-to-one correspondence between valid states of a system and attractors in the state space.

In a fully trained model, state transition trajectory connects all the attractors in the state space such that the state vector moves from one valid state / attractor to next on occurrence of a trigger.

During the training process, when the model is learning, it (the model) will do mistakes in state transitions especially in presence of noise i.e. invalid combinations of states and triggers in the training data. The attractors in state space are not well defined and the transition trajectory to is not well defined. Thru the error reduction mechanism, over multiple epochs, the model will learn the valid states / attractors and state transition sequence /trajectories and valid state, trigger combinations.

For this we first need to have a way of representing mistakes made during the training process and the way to do that is by defining an error function. At every state, the system can emit an output based on the predicted next state. If the output is not same as expected then the difference between expectation and output is an error. For e.g. when the FAWM is switched on (transitions from off to on state), the machine gives out a particular musical note indicating the change in state. Similarly, at the end of the wash cycle, the system goes back to idle state and while transitioning it emits a signature tune indicating it has transitioned to idle state. If the output is not what was expected, then it is an error in operation.

With the error function defined, the objective will be to minimise the error in each step. The learning will be inform of weights assigned to each of the dimension of the state space.

**Dynamic Systems, Attractors, Vector Fields and State Vector Transitions**

To understand the learning process and how the model learns to represent the states as attractors in state space, how the state transition happens on valid triggers, we need to imagine the state space to be filled with an invisible force field. Like a room in which air flows from the fan or ac vent in all directions and gets sucked out through sinks. Though we do not see the air flow, we can feel its force.

The attractors representing the valid states act as points in the state space where the forces converge with net force becoming zero. This is similar to a light floating objects that follow the air stream and get lodge in those parts of the room where the breeze is weak or nil. Note, state attractors can take the form of a ring in the state space (am going to keep this out of scope of this post).

The force field is such that the state vector (hinged to the origin), on occurrence of a trigger, is taken from one attractor to the next by the force field.

Pic 5

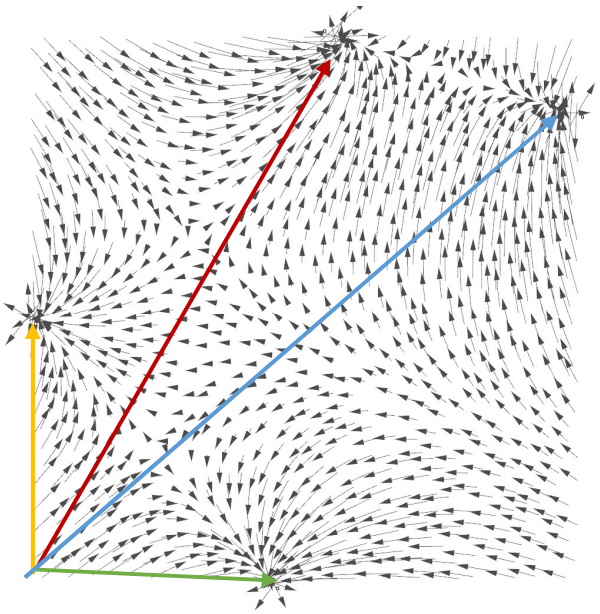


Image Source: https://generativelandscapes.wordpress.com/2014/08/21/interaction-between-vectors-example-6-3/

The force field can be represented by vectors / arrows at each point in the feature space representing the direction of the force at that point. That means there will be infinite arrows in the space as there are infinite points in the space. What we see in the picture above is a sample of these vectors. The length of the force field arrow indicates the magnitude of the force at that point (longer the arrow, larger the magnitude).

The colored arrows are the different state vectors. If the correct state transition is Orange to Red to Green to Blue, then the sate transition function NewState = f(Current State + Trigger) should somehow generate this force field to do the correct transition from one state /attractor to the other.

The logical question that comes to mind is where does this force field come from? To understand this, we need to step back and re-look at what is learnt during a model training phase. Be it Linear Regression, Neural Networks, Linear Classifiers or any other technique, all learning algorithms learn the relation between target and predictor variables thru a learning process. The lessons so learnt are expressed in form of coefficients or and represented as a matrix of weights. These weights capture the relationship between target and predictors as a ratio (dy/dx) called gradients.

**Gradients are Vector Fields**

When we build models to represent how target variables (represented by "y") is dependent on independent variables (represented as "X'), we are building "Associative Memory Model". The connection matrix "W", encodes the association between (X,y)

Let $y = Wx + C$

$dy/dx = d(Wx + C)/dx = ( (d(W)/dx) * x) + (W*d(x)/dx) + d(C)/dx = 0 + W + 0 = W$

W is a number with sign(indicating how much Y changes for a unit change in X). Number indicates the magnitude of change and sign indicates whether the change increase with increase of X or is it opposite. Combination of magnitude and direction make it a vector.

Thus, we can visualize the weights learnt by a model as a field of vectors in the feature space forming a force field! This field can further be imagined as a representation of the gradient at each point on an underlying surface that we call model (a.k.a. manifold).

**Pic 6**

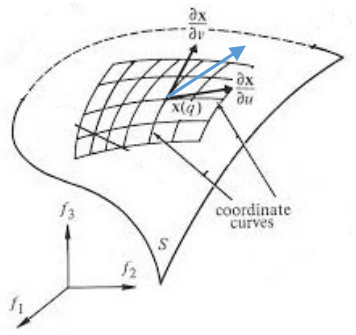
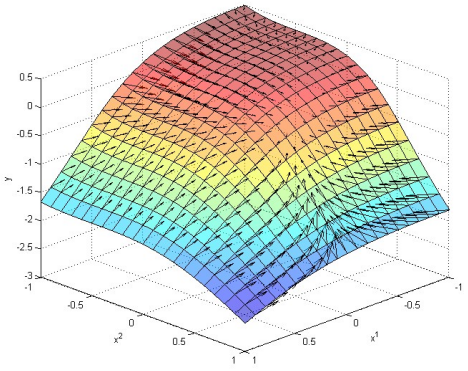



Image source: http://helper.ipam.ucla.edu/publications/bdcws1/bdcws1_15016.pdf

In the pic 6 above, the two dark arrows originating from a point in the curved grid represent the change in dimension f3 for unit change in the coordinates f1 and f2. Both are vectors and when added (vector addition), the resultant vector (blue) is the gradient (direction of increase and magnitude of increase) at that point. We can find the gradients at all the points (infinite) that make the surface (model). That will give us a field of infinite vectors. In the next picture we see a sample of vector fields with the underlying manifold.

**Pic 7**

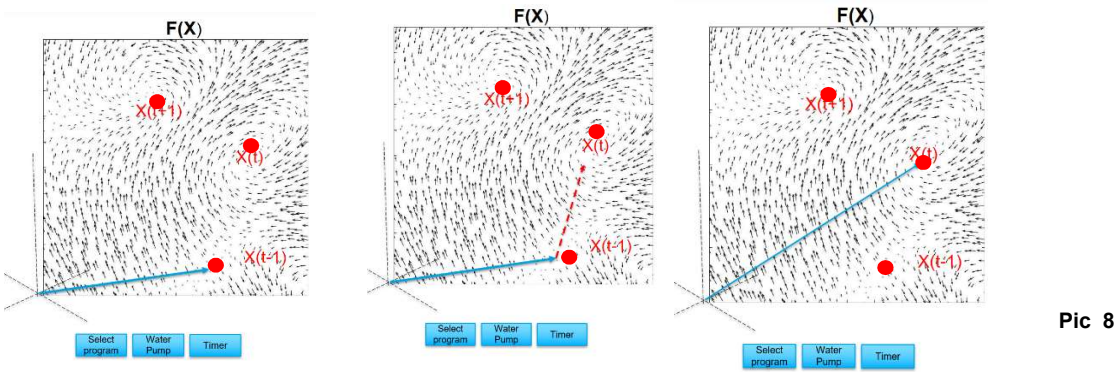


Source: https://www.researchgate.net/figure/Gradient-field-on-a-2-manifold_fig5_268572039

The locations where the surface is parallel to the ground, the gradient will be 0 i.e. the gradient vector will have 0 length and no direction (basically vanish). In the above image the region on the top will have very small vectors. These positions where the vector field becomes

0 are known as **attractors** in the states space and we can see that the vectors close to the attractor are converging towards the attractor. This region around the attractor is called the **basin of the attractor**. Imagine this vector field as an unseen force field that will push anything in that basin into the attractor like a marble rolling down the surface of a bowl.

To be technically accurate, we should be discussing the attractors and basins in terms of contour graphs (Ref: https://people.richland.edu/james/fall15/m221/projects/project11.html). Ignoring that for the time being, let us look at the state vector, trigger and the vector field interaction in the picture below.




**Pic 8**

In the pic 8, the left side starts with a state vector at x(t-1) representing "Programmed" state. The red dashed line is the "water pump switch on" trigger. The trigger vector too should be shown emanating from the origin. Instead, it is shown at the tip of the state vector after translation to represent vector addition leading to the next state. Thus, the trigger initiates state transition to "Programmed and Soak" state (Pic-8 3rd diagram). The "Timer" trigger will transition the WM to "Rinsing" state when the internal timer fires. The trigger has yet to fire and hence the "Rinsing" state is in future step (t+1).

If the vector field with attractors is difficult to grasp, think of it as the direction in which a marble will roll given the underlying surface with undulations. The red dots are the attractors which translate to locations on the surface where it is flat. When a trigger fires, the gradient vector fields i.e. the trajectory created by the orientation of the vector fields guide the marble from current location to the next.

In dynamical systems internal state (represented as state vector) evolves at every step (state transitions) but the vector field with the attractors remain stationery. Such systems are known as **autonomous systems**.

Pic 9

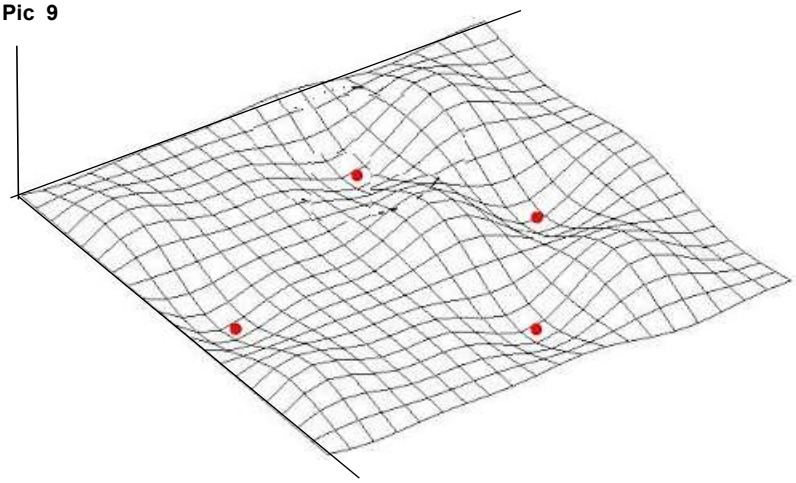


Image Source : http://www.scientificmythology.net/a7-WhatIsSciMyth.html

Instead of the vector field, pic 9 shows the underlying surface with four attractors (red dots) learnt thru training process. The behaviour of the system will be represented by state vector (not shown) transition from one red dot to next. The problem with this way of visualizing is depicting the transition trajectories. However, it is relatively more intuitive than the contour graphs.

In this discussion I have implied the attractors to be fixed points in the state space where the gradient vectors are 0. However, there are various types of attractors that can act as stable regions and which are not points. I have not discussed these here (Ref: "Computing with Attractors by John Hertz / "The Handbook of Brain Theory and Neural Networks").

One important point to keep in mind is that the states are internal to the system. They are not visible outside the system. That is why they are called internal states or hidden states. However, in every state the system can be made to emit an observable output (y). This output can be used to define error function. Minimising the error function will help define the vector field in which the attractors represent the internal states of the system.

**Formal Definition of State Machines**

Compared to functional systems in which the output is purely determined by input i.e. y is a function of X, in state machine output is determined by its history and current input. State machine models can be continuous in time or discrete. In this session we are going to study only the discrete state machines

If "t" is current time and "t+1" is the subsequent time unit in future then

1. **s(t+1) = f [ s(t) , i(t) ]** , state at time t+1 is a function of state at time t, the input trigger i(t). This expression represents the dummy data showed in pic 4
2. **y(t) = h [ s(t) ]** is the **observable output** of the machine at time t
    a. s – Internal state of the process
    b. i - Input at time t / trigger
    c. t - Time
    d. y – Output

e. h – Function to map current state to output y
f. f – Maps current state to future state at time t+1

Formal definition of state of a system - A set of quantities that summarize all the information about the past behavior of the system that is needed to uniquely describe future behavior.
Ref: "Adaptive Signal Processing"/ Handbook of brain theory & neural networks: Simon Haykin.

**Neural Networks for State Machine**

The washing machine example of state machine is a simple linear system. Systems we wish to model usually are more complex and non-linear in their behavior. A small change in input (triggers) lead to very different states. Thus, given a current state, there may be many possible trajectories leading to different possible states. The state transitions are probabilistic. To model such system, Neural networks turn out to be the most suitable methods in modeling.

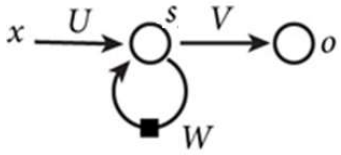
Neural networks are suited for the design of nonlinear state machines by virtue of three inherent characteristics

1. Ability to learn from representative training examples and generalizing
2. Nonlinear transformations thru Sigmoid or Tanh giving the ability to mimic non-linear behavior of physical processes
3. Universal function approximation capabilities due to which they can represent any mapping between the inputs and output to great degree of accuracy

However, conventional MLP is a static network. To mimic recurring nature of dynamic process, a feedback loop is introduced. Such neural network are called Recurrent Neural Networks (RNN).

**RNN for State Machines**

RNN is a variant of deep neural networks designed for creating sequential models i.e. models to capture the sequential dependencies between data points. Before we discuss RNN, we need to understand RNN Cell.




**RNN Cell**

Pic 10

Trigger X at time t multiplied by its weight U

Initial state S is state of the neuron at t-1

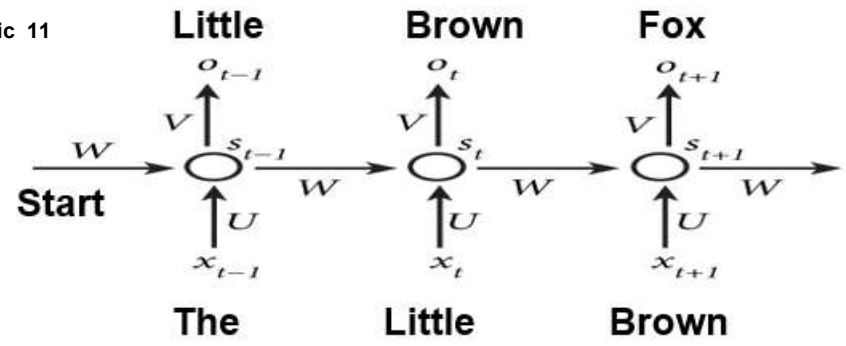
St+1 = Sigmoid( UX + WSt) (not shown)

Output "O" = Sigmoid( VSt+1)

Bias terms not shown for simplicity

1. RNN cells are different from the conventional neurons given the internal state that they maintain using which they can carry forward information from the past. RNN cells form the basic building blocks of recurrent networks.
2. Looks like feedforward neural network, but has connections pointing backward
3. When the input comes in a sequence, the internal state and the output keep changing
4. Note: **s(t+1) = f [ s(t) , i(t) ]** where the function f is represented by Sigmoid above
5. The hidden state of the RNN cell thus, is a record of previous states and trigger combination
6. Though RNN cell is one logical unit of the RNN, the cell in itself may be made of one or more neurons
7. The RNN when represented in time axis assuming every unit of time ** a new input from the sequence arrives, is called the unfolded RNN

Pic 11




8. Xt is input at time t, it could be a one-hot vector representing the word “little” while the previous word xt-1 could be one-hot vector representing “The”
9. St-1 is the internal / hidden state of the RNN Cell. It is the memory of the network. St is calculated based on previous state St-1 and new input xt.
10. Ot-1 is the output of the network at t-1 and could be probability distribution of next word from the vocabulary. Given the word “The” probability distribution of next words from the vocabulary. In this example, suppose “Little” has the highest probability

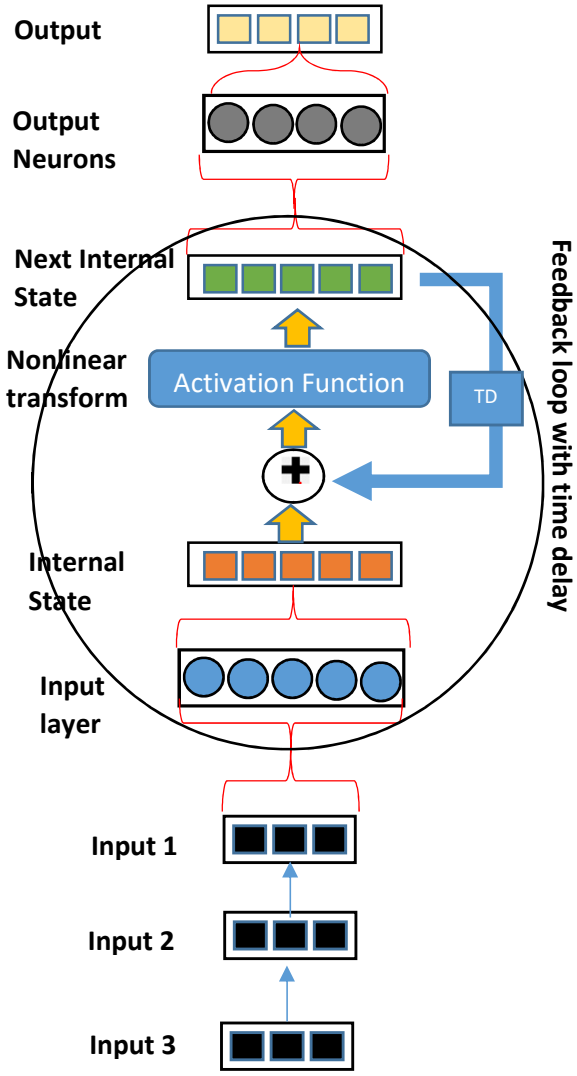
** Note: The time axis has been introduced to reflect the sequential nature of relation between the input data points. The input sequence may come in any frequency. Not necessarily at every tick of the clock.

**RNN Cell Internals**

To create a simple RNN cell in Tensorflow Keras, we use the following code-

```
model = Sequential()
step = 10
model.add(SimpleRNN(units=5, input_shape=(3,step), activation="relu"))
 # The output is fed back to the RNN automatically
```

In this code, a simple RNN cell with 5 neurons will be created. The input coming into these 5 neurons is of the shape ( 3, step ) where step = 3 i.e. input comes in a set of three sequential values.  This information can be represented pictorially as shown below –




Pic 12

**Simple RNN Cell Structure**

The encircled section is the RNN Cell. The composition of the cell includes –

1. Input neurons (five blue circles) that take input (three black boxes)
2. Vectorise the input (five orange boxes)
3. Add the input vector with previous internal state vector(five green boxes)elementwise
4. Nonlinear transform elementwise to generate next state(five green boxes)
5. The new internal state may be forwarded to output neurons (four dark circles) to generate an output (four light yellow boxes on the top)**
6. Output neurons (four dark circles) map the new internal state to predicted output(four yellow boxes)
7. Since this is a supervised learning algorithm, every input will be associated with expected output (not shown)
8. The predicted output will be used to estimate error (not shown) in predicted output
9. The error gradient will be back propagated to adjust the neuron weights during training step to minimize error
10. The weights thus learnt by input layer neurons represent the state space with vector field

** Note: It is not necessary for the RNN cell to emit an output at every iteration.

Note that the input is three dimensional (three black boxes per input), the vectorised form of the input (orange boxes) is five dimensional, the internal state (set of green boxes) is five dimensional and the output (set of yellow boxes) is four dimensional. All this is arbitrary to show that there is no dependencies between the input dimensions, internal state space dimensions and output dimensions.

The only requirement is that the vectorised form of the input should have as many elements as the internal state. This is because the vectorised input and internal state interact element wise. That is why in this picture both vectorised input and internal state have same number of elements or dimensions which is five

**Mathematical Structure of RNN Cell**

We can understand the structure in terms of the steps that an RNN Cell performs -

a. The input (X(t)) and the previous state (h(t-1)) are multiplied with respective weights Wih and Whh .
b. The two products are added elementwise and each resulting element is transformed using a non-linear activation function Fh
c. The result of this transformation is the next hidden state h(t)
d. The next hidden state h(t) is used to generate output y(t) at that time using an output function fo. This function takes weighted new state as input to generate the output
e. Thus, an RNN cell behaviour can be represented with the following two equations

$$h(t) = f_H\left(W_{IH}x(t) + W_{HH}h(t-1)\right)$$
$$y(t) = f_O\left(W_{HO}h(t)\right)$$

**Input Data Format**

Since this is a case of supervised learning, every step will involve a X_train, y_train combination. Thus if we are training RNN to predict words in a sequence such as "The little brown fox", the input data will look like –

| X_train | Y_train |
|---|---|
| Start | The |
| The | Little |
| Little | Brown |
| Brown | Fox |
| Fox | End |

Pic 13

The special key words "Start" and "Stop" are used internally to mark the beginning and end of an input sequence which is also known as an epoch (note: this is not the same epoch that we come across in training deep neural network).

## The Invisible Force Field

The set of weights that come into play in the state space (five dimensions in the picture above) together create the force field of gradient vectors.

Let us look at the flow of the data thru a pre-trained RNN Cell in form of a table shown below (Pic 14). The model is assumed to perform with 100% accuracy

| Input X_train | Vectorised X_train (of 5 elements) | Weighted Vector | Current internal state | Weighted current state | Next internal state | Generate output y_pred =Fo() | Y_pred / Next word | Y_train | Error |
|---|---|---|---|---|---|---|---|---|---|
| Start | 42596 | Wvect = W1*4, W2*2, W3*5,W4*9, W5*6 | H1 = h1,h2,h3, h4,h5 | mH1 = m1*h1, m2*h2, m3*h3, m4*h4, m5*h5 | NLF(mH1 + Wvect) = H2 = n1,n2,n3,n4, n5 | Fo (k1*n1, k2*n2, k3*n3, k4*n4, k5*n5) | The | The | 0 |
| The | 84627 | Wvect =W1*8, W2*4, W3*6,W4*2, W5*7 | H2 = n1,n2,n3, n4,n5 | mH2 = m1*n1, m2*n2, m3*n3, m4*n4, m5*n5 | NLF(mH2 + Wvect) = H3 = o1,o2,o3,o4, o5 | Fo (k1*o1, k2*o2, k3*o3, k4*o4, k5*o5) | Little | Little | 0 |
| Little | 56159 | Wvect = W1*5, W2*6, W3*1,W4*5, W5*9 | H3 = o1,o2,o3, o4,o5 | mH3 = m1*o1, m2*o2, m3*o3, m4*o4, m5*o5 | NLF(mH3 + Wvect) = q1,q2,q3,q4, q5 | Fo (k1*q1, k2*q2, k3*q3, k4*q4, k5*q5) | Brown | Brown | 0 |
| Brown | 19463 | Wvect = W1*1, W2*9, W3*4, W4*6, W5*3 | H4 = q1,q2,q3, q4,q5 | mH4 = m1*q1, m2*q2, m3*q3, m4*q4, m5*q5 | NLF(mH4 + Wvect) = c1,c2,c3,c4,c5 | Fo (k1*c1, k2*c2, k3*c3, k4*c4, k5*c5) | Fox | Fox | 0 |
| Fox | 89586 | Wvect = W1*8, W2*9, W3*5, W4*8, W5*6 | H5 = c1,c2,c3, c4,c5 | m1*c1, m2*c2, m3*c3, m4*c4, m5*c5 | NLF(H5 + Wvect) = j1,j2,j3,j4,j5 | Fo (k1*j1, k2*j2, k3*j3, k4*j4, k5*j5) | End | End | 0 |

**Pic 14**

The weights in the “Weighted Vector” and “Weighted Current State” columns together represent the different vector fields in the state space as shown below. The vector fields interact and lead to creation of the manifold with stable points called the attractors. Note the weights across instances are same!

Pic 15

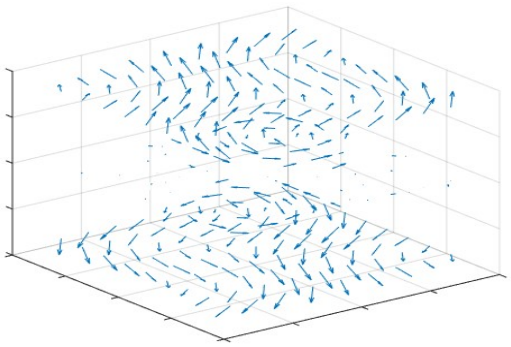
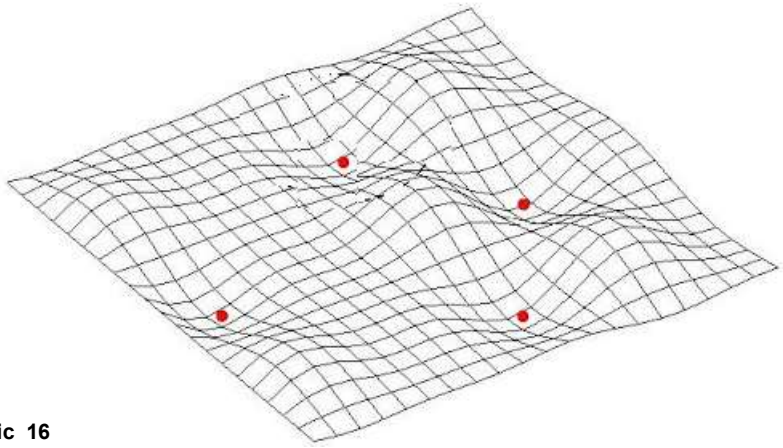


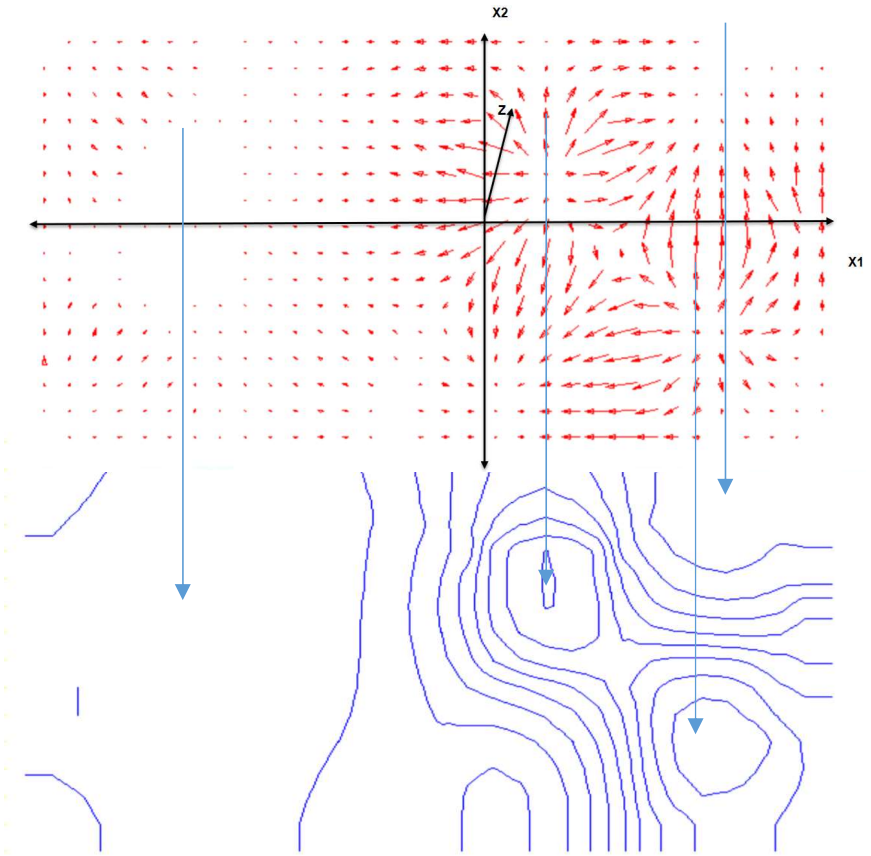
Image Source: https://serc.carleton.edu/details/images/106751.html

The two vector fields are shown with a gap for ease of understanding. Their total effect is a resultant vector field (not shown) that obtains the underlying manifold / surface shown below in the same state space.

Pic 16



At this point I would like to stress again that to be technically accurate, this discussion should be done using contour. However, the above figures are intuitive. An example is given below (pic 17)…




Pic 17

Mapping gradient field attractors to contour graphs

**Training RNN Cell**

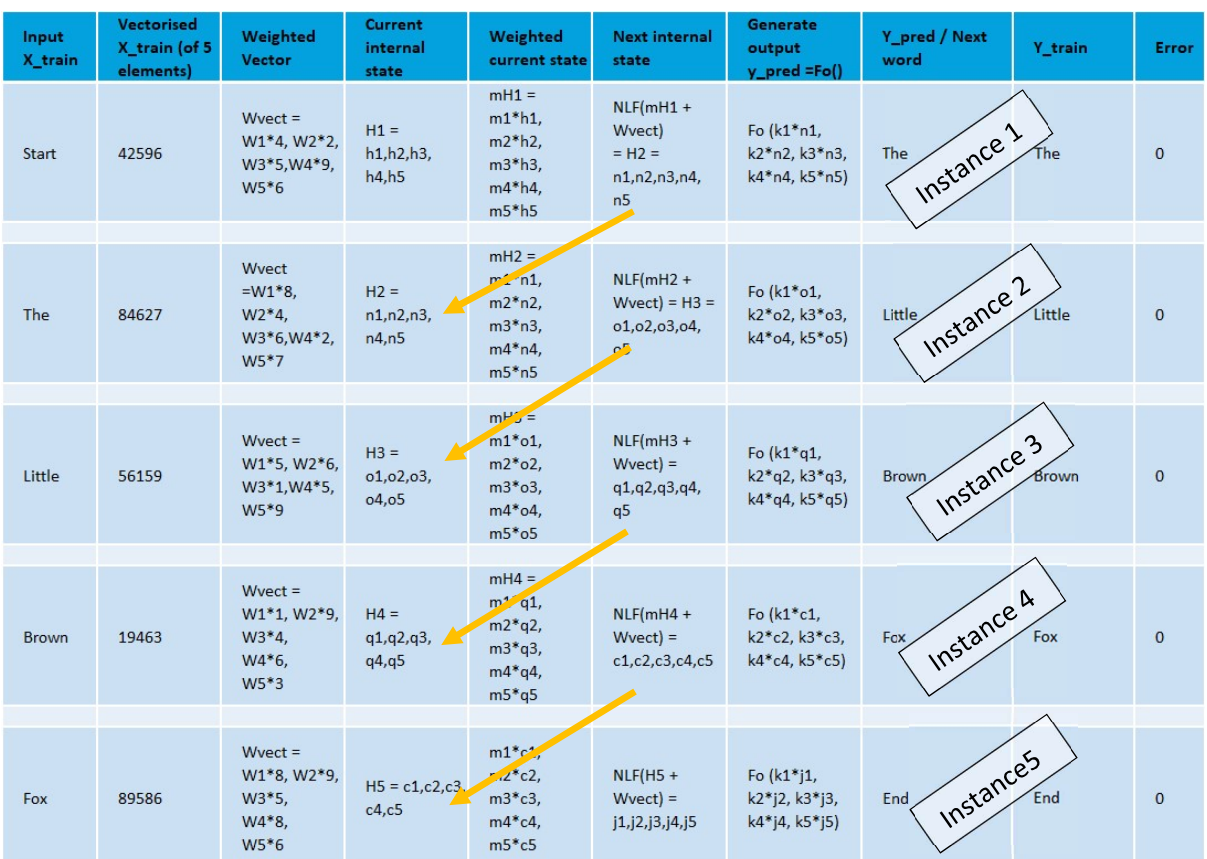
How are those weights adjusted to get the gradient fields with the internal states mapped to the attractors? To understand this, we need to understand unrolled RNN Cell (URC). URC is the same RNN Cell depicted at different points in the input sequence.Each instance of the URC is a snapshot of the RNN Cell with corresponding input, internal state and output.

Every instance of the URC has an input and expected output. It emits a predicted output and the difference between the expected and predicted output is the error that needs to be minimized. The complexity here is that error at a particular instance of URC depends not only on the input of that instance but also on all the preceding events. This is explained in the grid below.

## Unrolled RNN Cell

Let us look at the same grid we saw earlier for a fully trained RNN Cell.

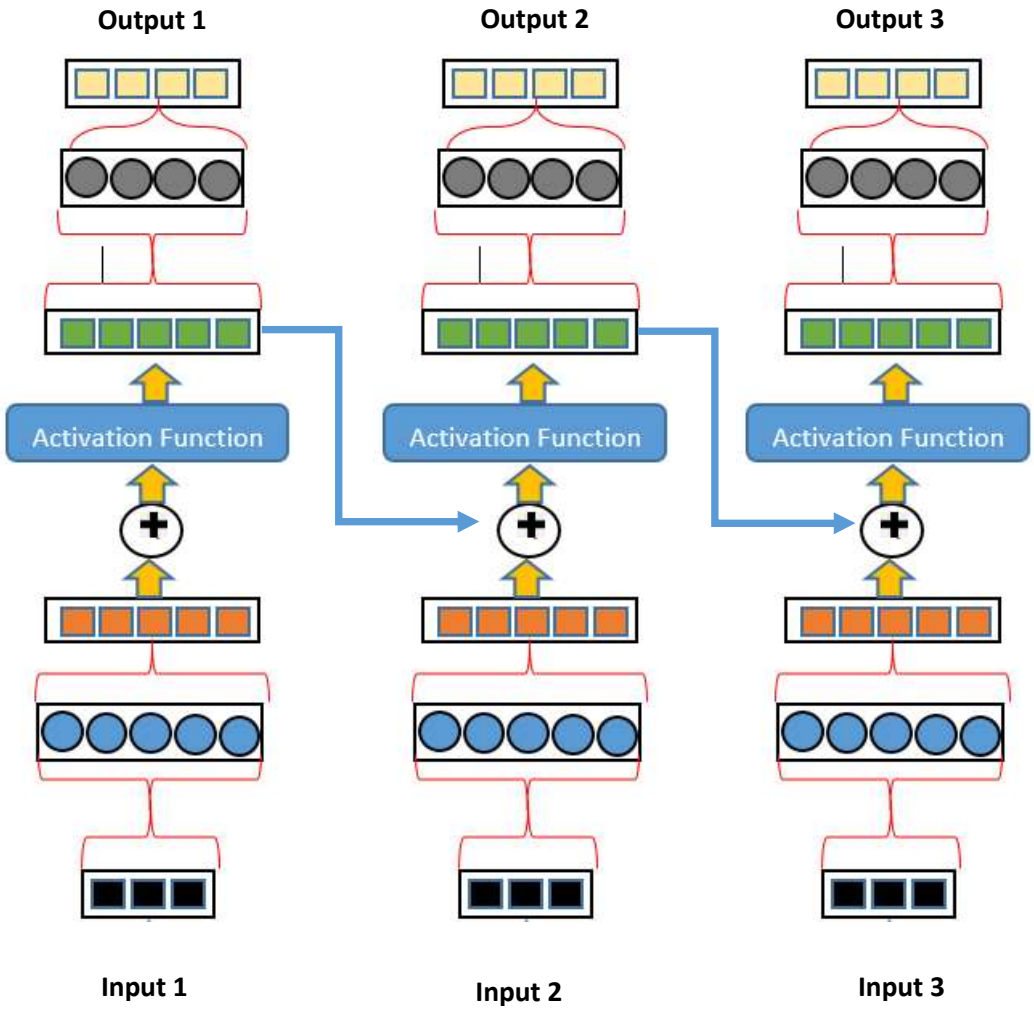
| Input X_train | Vectorised X_train (of 5 elements) | Weighted Vector | Current internal state | Weighted current state | Next internal state | Generate output y_pred =Fo() | Y_pred / Next word | Y_train | Error |
|---|---|---|---|---|---|---|---|---|---|
| Start | 42596 | Wvect = W1*4, W2*2, W3*5,W4*9, W5*6 | H1 = h1,h2,h3, h4,h5 | mH1 = m1*h1, m2*h2, m3*h3, m4*h4, m5*h5 | NLF(mH1 + Wvect) = H2 = n1,n2,n3,n4, n5 | Fo (k1*n1, k2*n2, k3*n3, k4*n4, k5*n5) | The | The | 0 |
| The | 84627 | Wvect =W1*8, W2*4, W3*6,W4*2, W5*7 | H2 = n1,n2,n3, n4,n5 | mH2 = m1*n1, m2*n2, m3*n3, m4*n4, m5*n5 | NLF(mH2 + Wvect) = H3 = o1,o2,o3,o4, o5 | Fo (k1*o1, k2*o2, k3*o3, k4*o4, k5*o5) | Little | Little | 0 |
| Little | 56159 | Wvect = W1*5, W2*6, W3*1,W4*5, W5*9 | H3 = o1,o2,o3, o4,o5 | mH3 = m1*o1, m2*o2, m3*o3, m4*o4, m5*o5 | NLF(mH3 + Wvect) = q1,q2,q3,q4, q5 | Fo (k1*q1, k2*q2, k3*q3, k4*q4, k5*q5) | Brown | Brown | 0 |
| Brown | 19463 | Wvect = W1*1, W2*9, W3*4, W4*6, W5*3 | H4 = q1,q2,q3, q4,q5 | mH4 = m1*q1, m2*q2, m3*q3, m4*q4, m5*q5 | NLF(mH4 + Wvect) = c1,c2,c3,c4,c5 | Fo (k1*c1, k2*c2, k3*c3, k4*c4, k5*c5) | Fox | Fox | 0 |
| Fox | 89586 | Wvect = W1*8, W2*9, W3*5, W4*8, W5*6 | H5 = c1,c2,c3, c4,c5 | m1*c1, m2*c2, m3*c3, m4*c4, m5*c5 | NLF(H5 + Wvect) = j1,j2,j3,j4,j5 | Fo (k1*j1, k2*j2, k3*j3, k4*j4, k5*j5) | End | End | 0 |




**Pic 18**

Each row in the grid in pic18 is one instance of the RNN Cell. The “Next Internal State” of the instance becomes the “Current Internal State” in the subsequent instance. This is shown by the arrows in the pic above.

The same information is often shown as a sequence of RNN Cells in a chain as shown below.




Pic 19

I have shown only three instances here due to space shortage. I have removed the labels too for the same reason. What flows from one instance to another (blue arrow) is the next internal state that becomes the current internal state in subsequent instance. In the addition operation, the internal state elements and the input vector (blue circles) elements are weighted and summed individually. The activation function acts on each element of the summation output to result into new state elements.

In the start of the training phase, all the weights are random weights and hence the outputs (predictions) will not be the same as the expected (target) and the difference will be error of each instance. Some things to be observed and kept in mind about the errors

1. Each instance predicts and errors
2. The error of an instance is influenced by the following
    a. weights applied to the input vector
    b. weights applied to internal state which depends on previous instance
    c. weights applied to new state to generate the output in a given instance
3. Using backpropagation, the weights associated with input vector and the internal state need to be adjusted to minimize error at instance and overall level

4. The weights across instances are the same weights i.e. the weights are shared across instances. In other words, the weights need to adjust in such a way that errors across all instances is minimized

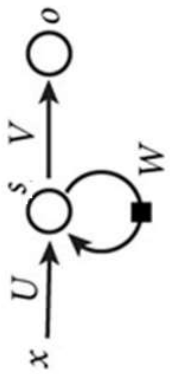
The following picture shows a rolled RNN Cell and its weight matrices. The weights in those three matrices should be learnt such that the error in all the iterations/ instances is minimized.

Wy

| C1 | C2 | C3 | C4 |
|---|---|---|---|
| W11 | W21 | W31 | W41 |
| W12 | W22 | W32 | W42 |
| W13 | W23 | W33 | W43 |
| W14 | W24 | W34 | W44 |
| w15 | w25 | w35 | w45 |

Wx

| A1 | A2 | A3 | A4 | A5 |
|---|---|---|---|---|
| W11 | W21 | W31 | W41 | W51 |
| W12 | W22 | W32 | W42 | W52 |
| W13 | W23 | W33 | W43 | W53 |
| W14 | W24 | W34 | W44 | W54 |
| W15 | W25 | W35 | W45 | W55 |




Wh

| B1 | B2 | B3 | B4 | B5 |
|---|---|---|---|---|
| W11 | W21 | W31 | W41 | W51 |
| W12 | W22 | W32 | W42 | W52 |
| W13 | W23 | W33 | W43 | W53 |
| W14 | W24 | W34 | W44 | W54 |
| W15 | W25 | W35 | W45 | W55 |

**Pic 20**

**Backpropagation Learning Process in RNNCell**

Let us take the three instances in pic19. Let the error done in each instance by L1, L2 & L3.

Total error by RNN is the sum of all the three errors and let this be Ltotal

$$L_1(y_1, \hat{y}_1) + L_2(y_2, \hat{y}_2) + L_3(y_3, \hat{y}_3) = L_{total}(y, \hat{y})$$

All the three weight matrices have to be updated to reduce $L_{total}$ i.e. $W_i := W_i - \eta \frac{\partial L_{total}(y, \hat{y})}{\partial W_i}$ for i = 1 to 3 for the three weight matrices.

Since the total error is a summation of three errors (belonging to three instances respectively), the weight adjustment will have to be done in such a way that error across all three instances reduce. That is achieved in the following way.

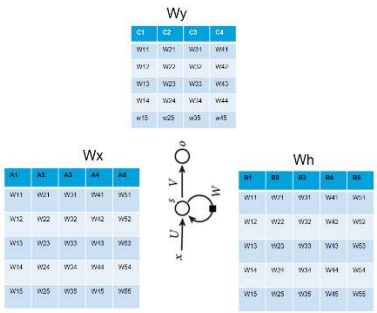
1. Let the number of neurons in the RNNCell be j which in our example is five (j = 5)
2. Let the number of instances be n which in our example is three i.e. n=3
3. The partial diff of error gradient for neuron j at instance n is given

$$\delta_{j,n} = \begin{cases} \varphi'(v_{j,n})\, e_{j,n} & \text{for } n = n_1 \\ \varphi'(v_{j,n}) \left[ e_{j,n} + \sum_{k \in \mathcal{A}} w_{jk} \delta_{k,n+1} \right] & \text{for } n_0 < n < n_1 \end{cases}$$

4. For last instance $n_l$ there is no subsequent instance. Hence error gradient is calculated with respect to the error of that instance alone.
5. For all other instances which is before the last instance, we need to account for error gradient which flows back from the subsequent instance. That is expressed as the plus term
6. This means the first instance now has to adjust weights for its error and also some portion on error from all subsequent instances
7. Thus, the chain equation for back propagation will have a component reflecting error gradient flowing in from the subsequent instances

**Chain Equation**

To understand the chain equation, let us keep the rolled RNN with weight matrices in view.




**Pic 21**

**<u>Chain equation with respect to weight matrix Wy.</u>**

Let the loss function be L and let $L_t$ be the loss at time t. The chain equation for error gradient with respect to Wy will be

$$\frac{\partial L_t}{\partial W_Y} = \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial z_t} \frac{\partial z_t}{\partial W_Y}$$

Where $Z_t$ represents the nonlinear transformation

Thus for the total error, it will take the form

$$\frac{\partial L_{total}}{\partial W_Y} = \frac{\partial L_1}{\partial \hat{y}_1}\frac{\partial \hat{y}_1}{\partial z_1}\frac{\partial z_1}{\partial W_Y} + \frac{\partial L_2}{\partial \hat{y}_2}\frac{\partial \hat{y}_2}{\partial z_2}\frac{\partial z_2}{\partial W_Y} + \frac{\partial L_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial z_3}\frac{\partial z_3}{\partial W_Y}$$

**Chain equation w.r.t the $W_x$**

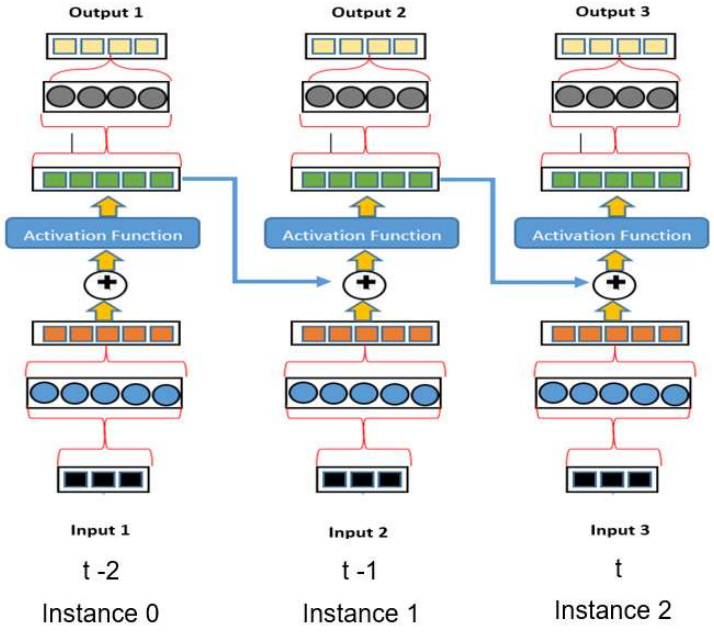
Loss gradient w.r.t. Wx matrix (weights associated with input vector) for current instance if it was the only instance -

$$\frac{\partial L_t}{\partial W_x} = \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial W_x}$$

The additional term $h_t$ is for the error gradient flowing thru the internal state calculation. But a given instance at step 't' is influenced by all the instances before it because it's internal state is influenced by all the previous states. In words, since $h_t = f(h_{t-1}, h_{t-2}, \ldots)$, this dependency too needs to be captured in the loss gradient w.r.t. Wx

$$\frac{\partial L_t}{\partial W_x} = \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial W_x} + \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial W_x} + \ldots + \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-n}}\frac{\partial h_{t-n}}{\partial W_x} = \sum_{k=0}^{n}\frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial W_x}$$

Look at the figure below where the last instance is the current instance at step 't'. The internal state at the third instance is influenced by states in all previous instances.




Pic 22

Now this same understanding has to be applied to all instances. That means when the instance "t-1" was the last/current instance, it was influenced by all preceding states.

But the equations is the loss gradient at individual instance. We want the loss gradient for total loss. For that we have to sum up all the loss gradients at individual instances. Thus for total loss w.r.t. $W_x$ will take the form

$$\frac{\partial L_{total}}{\partial W_X} = \sum_{t=1}^{n}\sum_{k=0}^{n}\frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial W_x}$$

Note the two summations. Outer summation is one less than inner because the very first instance has no instance before it. Therefore this loop starts from second instance (index =1). In pic 22 the second instance (in the middle) has a preceding instance hence, the inner summation starts from index K = 0.

**Chain equation with respect to weight matrix $W_h$**

The error gradient with respect to Wh is calculated in similar way as for Wx. It also has the same double summation. This is because of the same reason that at every instance the internal state is a function of all previous states.

Error gradient formula for with respect to Wh for current instance if it was the only instance –

$$\frac{\partial L_t}{\partial W_h} = \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial W_h}$$

Error gradient for loss function with respect to Wh at an instance in an unrolled RNN-

$$= \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial W_h} + \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial W_h} + \ldots + \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-n}}\frac{\partial h_{t-n}}{\partial W_h} = \sum_{k=0}^{n-1}\frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial W_h}$$

This chain will be formulated for every instance except the first one which does not have a preceding instance.

The total of all the error gradients w.r.t. for Wh will be summation of all instance gradients and will take the form –

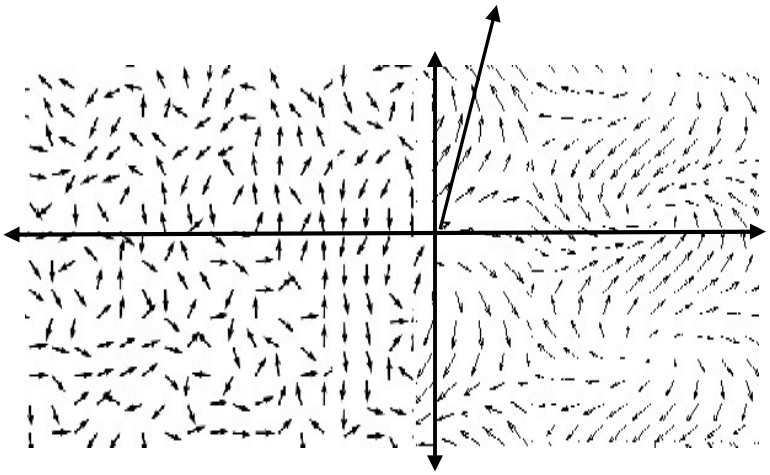
$$\frac{\partial L_{total}}{\partial W_h} = \sum_{t=1}^{n-1}\sum_{k=0}^{n}\frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial W_h}$$

Observe the outer summation starts from second instance (index = 1) while the inner summation starts from first instance (Index = 0).

**RNN Limitations**

Given the set of long chain equations, with so many parts in the equations, the learning process is computationally intensive. More the moving parts in a machinery, more likely a breakdown. Similarly, with so many calculations, even if a part of the chain equations collapse (vanishing gradient or exploding gradient), the RNN cell will not be able to learn the patterns i.e. associations between the target and independent variables and the sequence.

As a result of this, the RNN cell struggles to learn the patterns when the input sequences are long. Many authors refer this as inability of RNN learning the long term dependencies. In pictorial way, the gradient vector field fails to crystallize into appropriate shape where the internal states are mapped to attractors.



Pic 23

Some parts of the vector field remain ill formed while other regions start shaping up into attractors. This happens when the weight matrices are not update taking into account the error gradients from the entire sequence preceding the given instance.

Thus for the initial instances where the equations are not long, the vector field maps the internal states to attractors but for instances coming later in the unrolled RNN, given the length of the chain, it fails map the state to attractors.

This limitation of RNN in handling long sequences, was overcome using another technology called LSTM (Long Short Term Memory) which is another variant of Deep Neural Network for sequential learning.