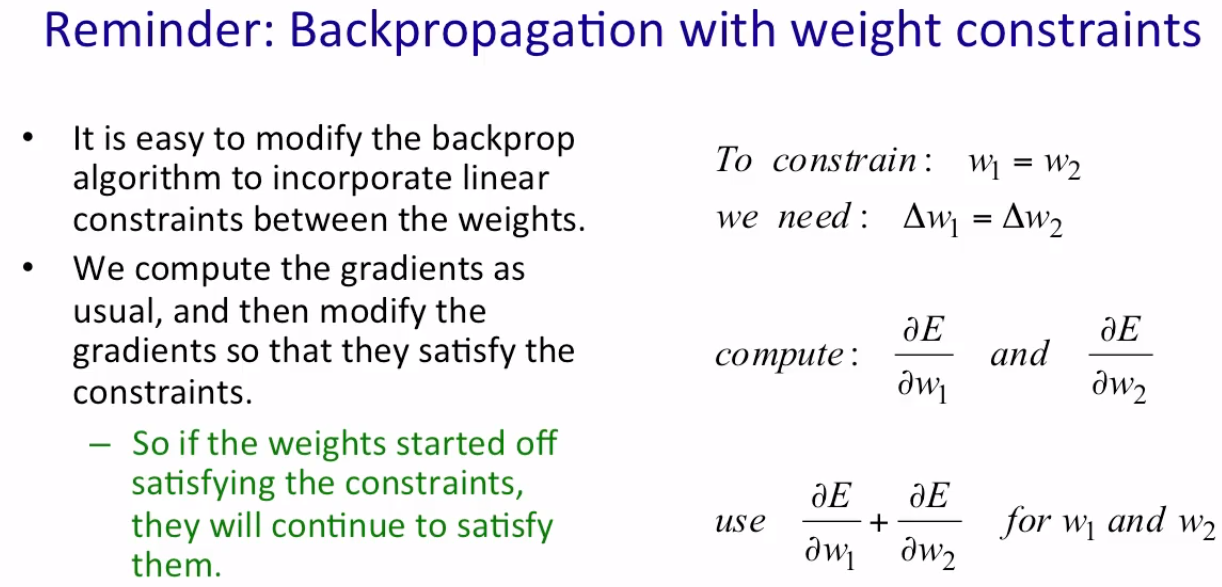
Recurrent Neural Network

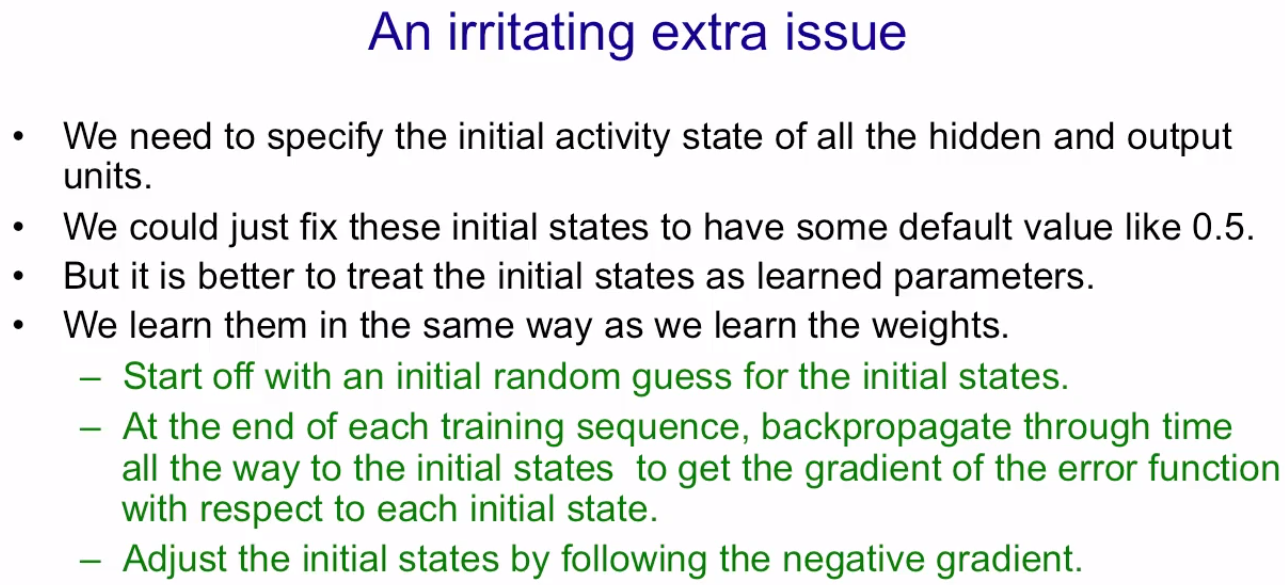
# Backpropagation through time

It is the same as a feedforward network that has been expanded in time. It is a layered feedforward network where the weights are constrained to be the same at every layer (point in time).

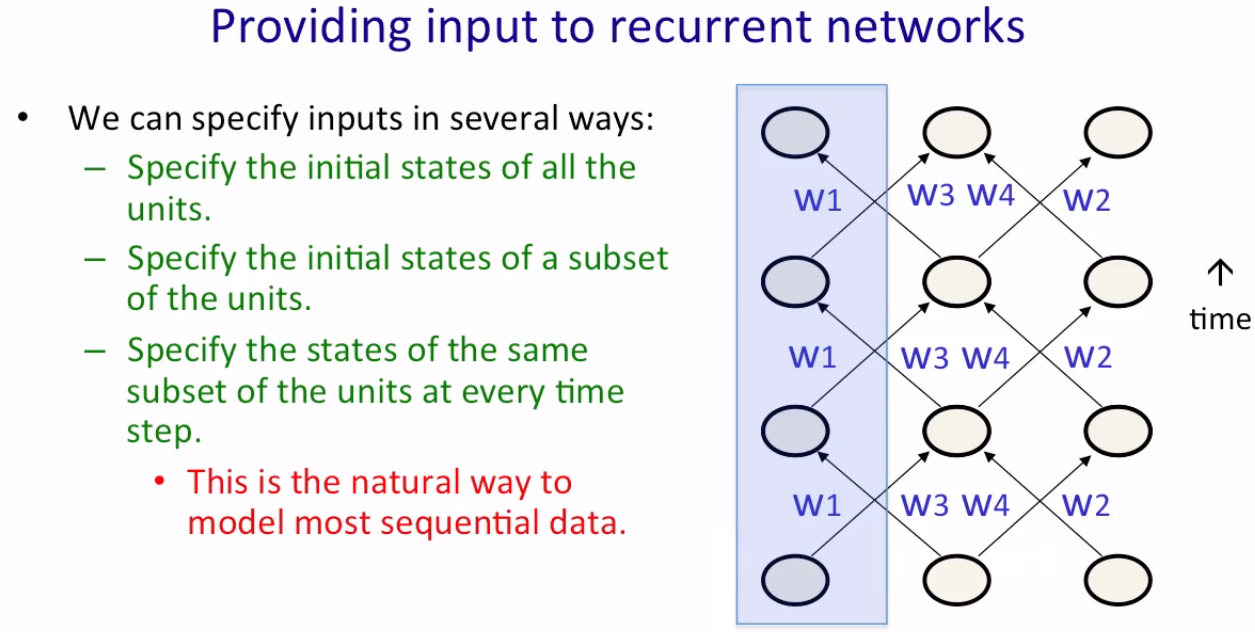


## How to change weights

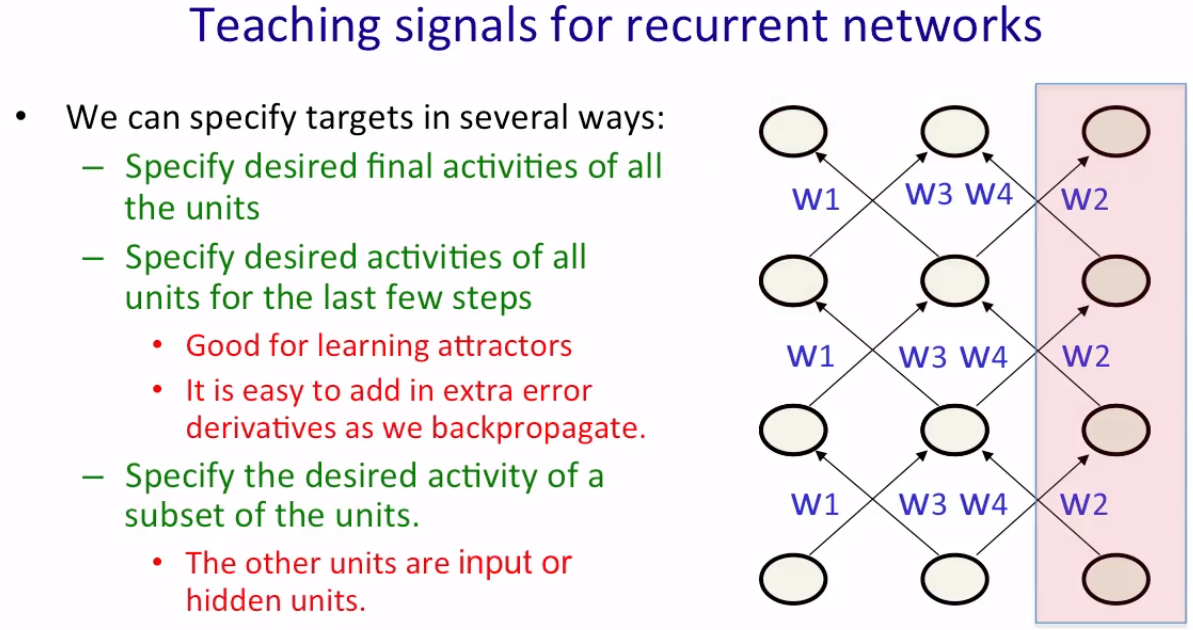
* The forward pass builds up a stack of activities at each time slice.
* The backward pass peels activities off that stack and computes error derivatives each time step backwards. That's why it's called back propagation through time.
* **Weight calculation**:
  + After the backward pass we can add together the derivatives at all the different time step for each particular weight.
  + Then change all the copies of that weight by the same amount which is proportional to the sum or average of all those derivatives.



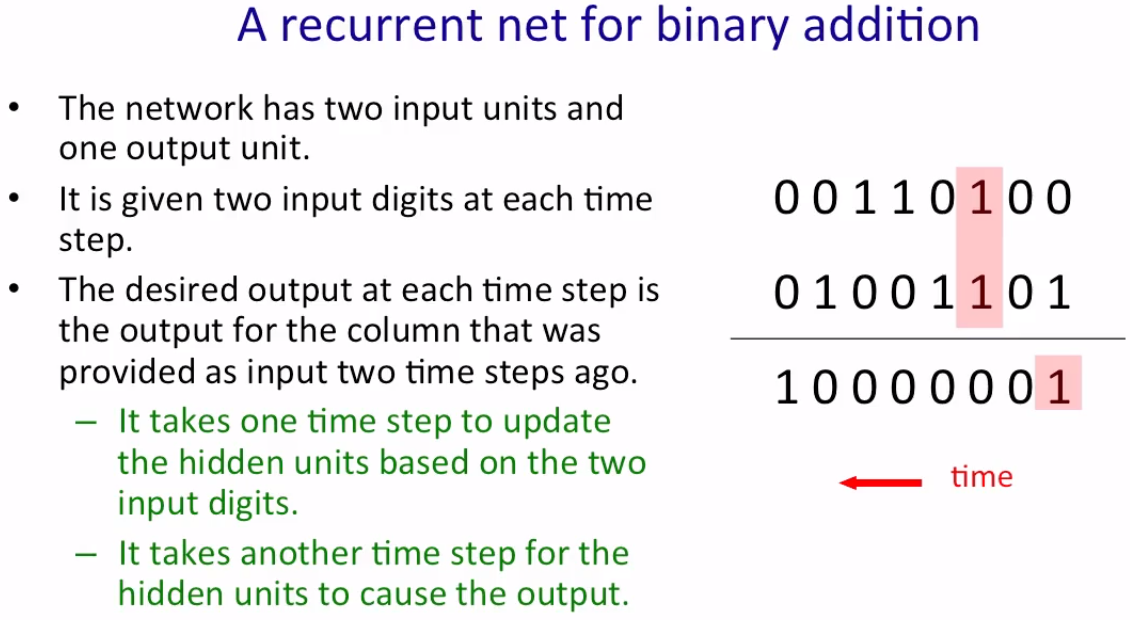
## How to provide inputs



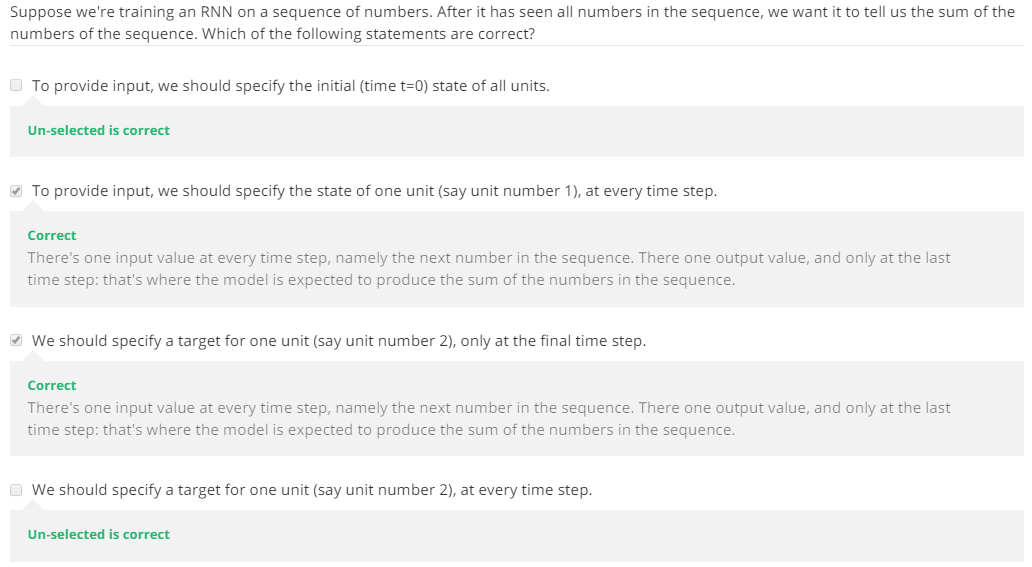
## How to provide targets



## Example



## Trivia



* How many bits do you require to represent a given number of possibilities? - Shannon’s theorem
* How many possibilities (in bits) can you represent by a given number of bits?

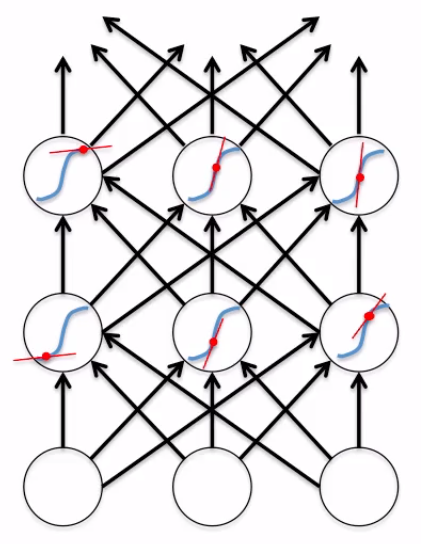
(N is the number of bits)

# Training a recurrent network

## Difficulties

### Exploding and vanishing gradients

Forwards passes are logistic. They restrict the gradients or activity vectors inside the range [-1,1]. However, backward passes are linear. So, for example, doubling the error derivative in the final layer would double all of the error derivatives.



### Problem with linear system

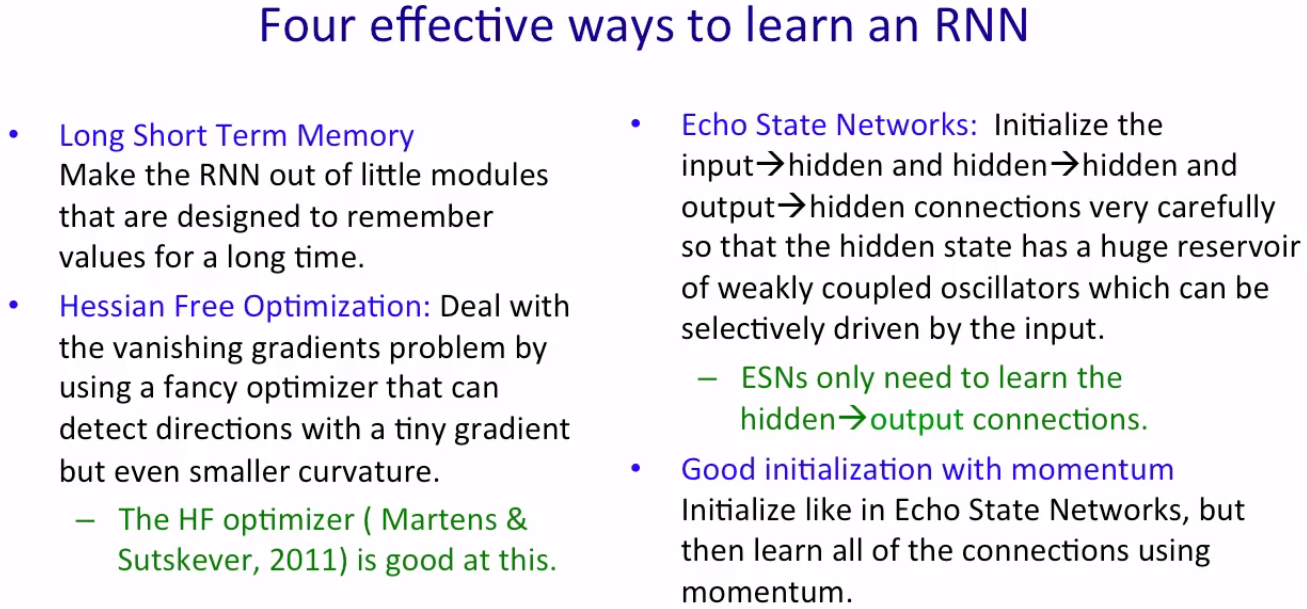
When we iterate through a linear system, the gradients either explode or die. Therefore, iterating through time, in the context of an RNN, will make the gradients either huge or tiny by the time they reach the target and wipe out all the knowledge/memory.

Gradient’s value, that tends to explode or die, is proportional to the power of the number of times it iterates through time. For example, if we have a recurrent neural network trained on a long sequence, for example 100 time steps, then if the gradients are growing as we back propagate, we'll get whatever that growth rate is to the power of 100 and if they're dying, we'll get whatever that decay is to the power of 100.

### Conclusion

RNNs have difficulty dealing with long-range dependencies.

## How to deal with the difficulties



# Format of input data is important

[**James Blaha**](https://disqus.com/by/jamesblaha/) [RalfD](http://karpathy.github.io/2015/05/21/rnn-effectiveness/#comment-2038962893) • [2 years ago](http://karpathy.github.io/2015/05/21/rnn-effectiveness/#comment-2057322743)

I fed in famous guitar tabs (about 500mb) worth, in ASCII. It now generates guitar tabs in ASCII well enough for me to import them into GuitarPro, where I recorded it playing back. I took that and imported to FL Studio, added some filters and a drum loop, but the notes and rhythms are otherwise totally unedited. This is what it sounds like:

[https://soundcloud.com/opto...](https://disq.us/url?url=https%3A%2F%2Fsoundcloud.com%2Foptometrist-prime%2Frecurrence-music-written-by-a-recurrent-neural-network%3AqfsHnCPDgpP1Wn3xSO-LwiR2ZEw&cuid=3095056)

It is only about 20% done training on the file, and already getting good results!



[**bmilde**](https://disqus.com/by/bmilde/) [James Blaha](http://karpathy.github.io/2015/05/21/rnn-effectiveness/#comment-2057322743) • [2 years ago](http://karpathy.github.io/2015/05/21/rnn-effectiveness/#comment-2112426109)

Hi James!



[**James Blaha**](https://disqus.com/by/jamesblaha/) [bmilde](http://karpathy.github.io/2015/05/21/rnn-effectiveness/#comment-2112426109) • [2 years ago](http://karpathy.github.io/2015/05/21/rnn-effectiveness/#comment-2127591563)

Hi! I'd be happy to! Here is a bunch of stuff from it, including the dataset, trained models, and the files I used to convert the tabs back and forth.

[https://drive.google.com/fo...](https://disq.us/url?url=https%3A%2F%2Fdrive.google.com%2Ffolderview%3Fid%3D0BxIbIVKS-qnNfmhuUDkyNUdIaHlBOHBuSG4yS215cGtKNkZ0NEtZWi1oYUVWOU8xT3VpUXM%26usp%3Dsharing%3AmHAsBQWIDyXaozkqyalwWijcJEo&cuid=3095056)



[**karpathy**](https://disqus.com/by/karpathy/) **Mod** [James Blaha](http://karpathy.github.io/2015/05/21/rnn-effectiveness/#comment-2057322743) • [2 years ago](http://karpathy.github.io/2015/05/21/rnn-effectiveness/#comment-2057462799)

Neat, this is fun! What format is the input in?

[**James Blaha**](https://disqus.com/by/jamesblaha/) [karpathy](http://karpathy.github.io/2015/05/21/rnn-effectiveness/#comment-2057462799) • [2 years ago](http://karpathy.github.io/2015/05/21/rnn-effectiveness/#comment-2057576798)

I formatted the input like:

%  
E|------------------------------|--------------------------------------------------|  
B|------------------------------|--------------------------------------------------|  
G|------14--12--12--9---9---7---|-------5--------5---------------------------------|  
D|------14--12--12--9---9---7---|----3--------3--------5--7-----5--7--5--7--5------|  
A|------------------------------|-3--------3--------5--------5-----------------7---|  
E|-0----------------------------|--------------------------------------------------|  
%  
E|--------------------------------|-------------------------------------|---------------------------|  
B|--------------------------------|-------------------------------------|---------------------------|  
G|--------------------------------|-5-----------7---7--7--0-------------|---------------------------|  
D|------5---7---7---5---7--5------|-5----0------7---7--7----------------|---------------------------|  
A|----------------------------7---|-3-----------5---5---------7--5------|------5---7---7---5---7----|  
E|-0------------------------------|---------------------------------7---|-0-------------------------|  
%

(Everything lines up with a monospace font) Standard ASCII tab format. I exported like this but removed all lyrics/comments and added the % between tab lines to help it differentiate new lines of tabs.

It gives me output that is much more consistent than the input:

%  
E|-0-----------0-----------0-------|-0-------------------0-----------|-0-------------------------------|  
B|---------1-----------------------|---------1-----------1-----------|---------0-----------0-----------|  
G|-----2-----------2-----------2---|-----0---------------0-----------|-----0---------------0-----------|  
D|-2-----------0-------0-----------|---------------------------------|---------------------------------|  
A|---------------------------------|---------------------------------|---------------------------------|  
E|---------------------------------|---------------------------------|---------------------------------|  
%  
E|---------------------------------|---------------------------------|---------------------------------|  
B|-------------0-------------------|-------------0-------------------|-------------0-------------------|  
G|-----0-----------0-----------0---|-----0-----------0-----------0---|-----0-----------0-------0-------|  
D|---------------------------------|---------------------------------|-----------------0---------------|  
A|---------------------------------|---------------------------------|---------------------------------|  
E|---------------------------------|---------------------------------|---------------------------------|  
%

This is what i put back into GuitarPro and have it play, usually only needs minor fixes. Sometimes it likes to add another empty bar on just one line, or remove one, for instance.

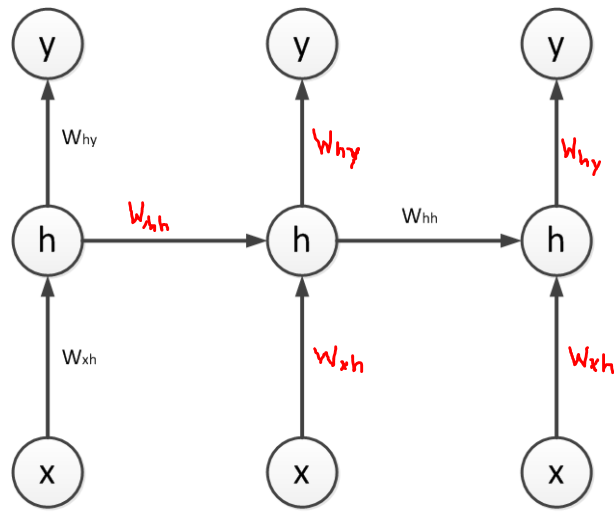


[**karpathy**](https://disqus.com/by/karpathy/) **Mod** [James Blaha](http://karpathy.github.io/2015/05/21/rnn-effectiveness/#comment-2057576798) • [2 years ago](http://karpathy.github.io/2015/05/21/rnn-effectiveness/#comment-2057579226)

hmm I'm not sure that this is very RNN friendly input form. Wouldn't it be better if you gave it contiguous chunks in time? E.g. instead of   
E 1 2 3  
B 4 5 6  
G 7 8 9  
you'd pass in something like 123.456.789. In other words, you're passing in groups of 6 things that all happened at the same time, and they are always delimited with a special character such as the dot.

I'd expect that to work significantly better

# Quiz



T = 0 T = 1 T = 2

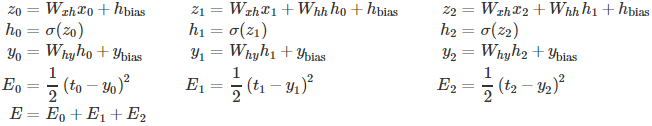
The figure above shows a Recurrent Neural Network (RNN) with one input unit *x*, one logistic hidden unit *h*, and one linear output unit *y*. The RNN is unrolled in time for T=0, 1, and 2.

The network parameters are: *Wxh* = −0.1, *Whh* = 0.5 , *Why* = 0.25 , *h*bias = 0.4, and *y*bias = 0.0.

If the input *x* takes the values 18, 9, −8 at time steps 0, 1, 2 respectively, the hidden unit values will be 0.2, 0.4, 0.8 and the output unit values will be 0.05, 0.1, 0.2 (you can check these values as an exercise).

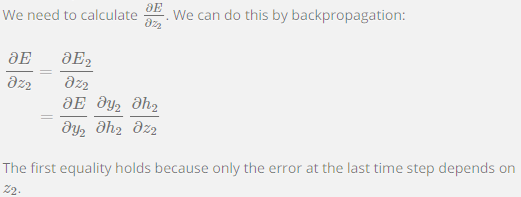
A variable *z* is defined as the total input to the hidden unit before the logistic nonlinearity.

If we are using the squared loss, with targets *t*0, *t*1, *t*2, then the sequence of calculations required to compute the total error *E* is as follows:



If the target output values are *t*0=0.1, *t*1=−0.1, *t*2=−0.2 and the squared error loss is used, what is the value of the error derivative just before the hidden unit nonlinearity at *T*=2 (i.e. )? Write your answer up to at least the fourth decimal place. – Hint in next page

Hint:



# Miscellaneous

## Rank of a matrix

The maximum number of linearly independent rows in a matrix A is called the **row rank** of A, and the maximum number of linearly independent columns in A is called the **column rank** of A. For example, the rank of a 3 x 5 matrix can be no more than 3, and the rank of a 4 x 2 matrix can be no more than 2. In short, rank of a matrix is usually the minimum of the two:

In plain language, you find any row/column that cannot be generated by performing any mathematical operation on the rest of the row/column respectively. The minimum of these two numbers is the rank of a matrix.

### How to calculate

* 1. Transform the rows into a reduced row echelon form, i.e. transform one non-zero element to a zero and apply the same computation for the rest of the elements in the row.
     + Achieve this by adding/subtracting/multiplying (element-wise) this row with any other row(s?) in the matrix.
  2. Find the number of pivot rows after the step above is done. – all zeros except for one 1.
  3. The number of pivot rows is your **row rank of the matrix**.
  4. Do the same for the columns to get the **column rank of the matrix**.
  5. Take the minimum of ‘c’ and ‘d’. This is your **rank of the matrix**.

### Takeout/application

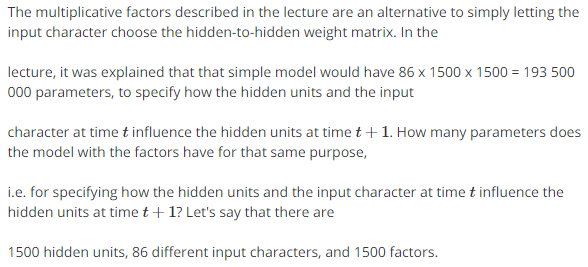
This will give you an idea of how the (row/column) vectors in the matrix are co-related (??), i.e. if, in fact, any of them are related and if so, how many. This provides an overview of the inter-dependencies among the elements of the matrix.

reference: [Khan Academy - rank of a matrix](https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/null-column-space/v/dimension-of-the-column-space-or-rank)

# Multiplicative model

Reduces the dimensionality of a typical, **additive RNN structure**, sometimes to the factor of 60 or above! Mostly used for character prediction in a string to generate comprehensive corpuses, such as Wikipedia articles.

In a 2-layer model (time and ), each factor has three different connections: 1 coming from the input units, one coming from the hidden units at time and one coming from the hidden units at time



In the additive (simpler) model, all input units/characters provide inputs to all the factors. This can make the network grow very large. On the other hand, **the multiplicative model is an alternative to simply letting the input character choose the hidden-to-hidden weight matrix**.

# Echo State Network (ESN)

* The hidden-to-hidden connections are not learned, and without learning, the hidden layer does not contribute to overfitting.
* ESN's don't backpropagate through time, so gradients do not matter here. In other words, ESNs do not suffer from exploding or vanishing gradients unlike RNNs.

# A thorough explanation of Factors

**A Thorough Explanation of Factors**

Kyle Hipke[Week 8](https://www.coursera.org/learn/neural-networks/discussions/weeks/8) · [a month ago](https://www.coursera.org/learn/neural-networks/discussions/weeks/8/threads/wKeYgKYhEeeF0Qp3MGZafg) · Edited

I'm struggling with understanding factors, so here's my best shot at explaining it better.

Some background to start with - one of the original papers that describes factors is [here](http://www.cs.utoronto.ca/~ilya/pubs/2011/LANG-RNN.pdf). In that paper, they describe what they call an MRNN - multiplicative recursive neural net, which is an RNN that's implemented using factors. That's what Hinton's lecture tries to describe.

To start, let's just ignore the equation that shows c as a summation. I think that equation is likely to cause confusion (or I have just failed to properly understand factors). Instead, we'll look at the equation that shows cf = (bT\*wf)(vf\*ufT)a. Note that the equation in the slides is wrong - they need to make the (uf\*vfT) instead be (vf\*ufT)

First, I'll talk about the case of having **just one factor**. And I'll use the example of an actual MRNN rather than just talking about groups a, b, and c.

b is the activations of "group b", which in the MRNN example is just the activations of the input neurons. So if there's 6 inputs it's a vector of length 6. b would be a 6x1 matrix.

wf are the weights of the connections from the input neurons to the factor. This is a vector that's the same length as b since there's one weight per input neuron. This is unlike a normal RNN where each input neuron can connect to more than one hidden neuron. In that case, we would have to represent the weights using some 2 d matrix so it could model the weights from each input neuron to each hidden neuron. But in this case, all of the inputs connect directly to a single factor. So if there are 6 inputs, wf is 6x1.

bT\*wf is therefore just a scalar - a single number - the result of multiplying the vectors (1x6 \* 6x1 = 1x1). Just in case it isn't clear - bT is just the transpose of b, swapping the rows and columns in order to make the multiplication work (i.e. it turns the 1xn vector into a nx1 vector) - the result is that each element of b is multiplied by the corresponding element of wf and all of those are added together to get the scalar). b1\*wf1 + b2\*wf2 ... = the scalar.

uf are the weights of the connections from group a (the hidden units at time t) to the factor. This is a vector of length (number of hidden units), because each hidden unit connects to the single factor. So if there's 100 hidden units, uf has length 100. Uf would be a 100x1 matrix.

vf are the weights of the connections from the factor's output to group c (the hidden units at time t+1). This is ALSO a vector of length (number of hidden units) because the factor connects to every hidden unit. So if there's 100 hidden units, vf will be of length 100. Vf would be a 100x1 matrix.

So, uf\*vfT - don't make the mistake of thinking this is a scalar. If there's 100 hidden units, this will be multiplying dimensions 100x1 \* 1x100 which gives a result that has dimensions 100x100.

a is the activations of group a, which in the RNN is the activations of the hidden units at time t. This is a vector of length (number of hidden units) because there's one activity value per hidden unit. So if there's 100 hidden units, a will be 100x1.

Finally, cf is the inputs to group c sent by the factor. In the MRNN, group C is the hidden units at time t+1. So cf is also a vector of length (number of hidden units) which describes the inputs sent to the hidden units at time t+1. If there's 100 hidden units, cf has length 100 and is a vector of dimensions 100x1. Cf is the result of the factor - it describes what values are sent to the hidden units at time t+1. Remember that these values, cf, are the ONLY inputs that the hidden units receive. The hidden neurons are NOT getting input from anything else.

Keep in mind 2 key things - cf is NOT the actual ACTIVITIES of the hidden units at time t+1. It's just the value that is SENT to them at that time. Furthermore, cf represents the value AFTER being multiplied by the hidden unit weights, so cf will NOT be multiplied by any more weights. It is akin to the "z" value for a neuron - the weighted sum of all inputs. There are no parameters that are learned for the hidden units other than what's in the factors. There are no "extra weights" that are used on cf after it has been calculated in order to get the final activities. In other words, once cf has been calculated, all that's needed in order to calculate the activities of the hidden units in the next time step is to (if they are logistic units) take the log of the values in cf.

So, when you look at the case of a single factor, there's just one "loop" of that summation you posted. The bT\*wf is a scalar and uf\*vfT is a 100x100 matrix. The scalar multiplies that matrix. Now you're left with a 100x100 matrix and the vector a, a 100x1 vector. When you multiply that 100x100 matrix with the 100x1 vector, you are left with a 100x1 vector, which is c. In other words, **the factor uses the values that have been set for uf, wf, and vf along with the activities of the input neurons, in order to define a 100x100 matrix. The hidden neuron activities at time t are multiplied by this 100x100 matrix and sent as the inputs to the hidden neurons at time t+1.** The 100x100 matrix is called a "transition matrix" because, when multiplied by a 100x1 vector, it produces a new 100x1 vector where each of the individual elements in the new vector are based on all of the values in the original 100x1 vector.

More concretely, when you do the actual matrix multiply of the 100x100 transition matrix times the 100x1 vector, each element in the result vector is calculated by multiplying each element in the input vector by the values in one row in the transition matrix and adding those multiplies all together. So you can think of an individual row in the transition matrix as being the "weights". To calculate the first element of the result - Row 1, column 1 of the transition matrix will be multiplied by vector element 1, and row 1, column 2 of the transition matrix will be multiplied by vector column 2, and so on for the entire first row. Then all the products will be added together to get a single scalar value. That value will be the first element of the result vector.

So, think about a factor in this way - **Let's say we have n hidden units. To calculate the values sent to the hidden units at the next time step, the input neuron activities and all of the factor parameters (vf, wf, and uf) are used to generate a transition matrix of size n x n. The activities of the hidden units at the current time step (an nx1 vector) are multiplied by the transition matrix to produce an nx1 vector representing the values sent to each hidden unit at the next time step.**

So given all that, what actually IS a factor? What defines a factor? Simple - it's defined by those 3 vectors - uf, vf, and wf. Uf and vf are of length (number of hidden units) and wf is of length (number of input neurons). So the number of parameters in one factor is (# hidden units)\*2 + (# of inputs). Each of the values in those vectors are learnable parameters. Note that these parameters REPLACE the weight parameters that hidden units normally learn. The idea is that we are using factors so we can make the weights between hidden units change based on the inputs. So we don't want to still learn extra weights for each of the inputs that are coming from the factors. We just want the factors to BE the weights for the hidden-hidden connections.

Now let's consider the case for **multiple factors**. For example, an MRNN with 100 factors. For each of those factors, you'll have to separately define parameters wf, uf, and vf. I was originally confused in this case because I thought the following - each hidden unit now has 100 ouputs - 1 going to each factor. Similarly, every hidden unit now has 100 inputs - one coming from each factor. And then it learns a weight for each of those 100 inputs, on top of all the parameters that were learned for each factor. THIS IS WRONG! Each hidden unit actually does have one output to each factor (so, 100 total), but it still ONLY has 1 input coming to it, TOTAL, REGARDLESS of the number of factors (which, if it's logistic, it could simply take the log of that input to get its new activity). And it DOES NOT learn a weight for that value - the weights were already taken care of inside the factors.

As for understanding that summation you posted - Cf is the inputs to the hidden units from ONE FACTOR. C is the inputs to the hidden units resulting from ALL FACTORS. Say we have 100 factors. Instead of computing all of the 100 transition matrix \* input vector multiplies separately and sending each of those resulting vectors as inputs to the hidden units at the next timestep, we can just calculate the transition matrix for each factor (based on the factor parameters and input neuron activities) and then ADD those transition matrixes all together to get a new, FINAL transition matrix. Then to calculate the values sent to the hidden units at the next time step, we only need to do 1 multiplication - multiply the FINAL transition matrix by the activities of the hidden unit at the current time step. So, regardless of how many factors you have, you'll, in the end, end up with a single transition matrix that defines how the current hidden activities should be used to determine the input to the hidden units at the next time step. In order for this hidden unit to calculate its next activity, all it needs to do is (for a logistic unit) take the log of its input value. Remember, there are no weights on top of what has been sent in c to the hidden unit.