Chapter – 7

**Neural Networks for Machine Learning**

**Geoffrey Hinton**

with **Nitish Srivastava** & **Kevin Swersky**

**RNNs**

Lectures: Geoffrey Hinton

Modeling Sequences a brief overview

Training RNNs with Back Propagation

A toy example of Training an RNN

Why it is Difficult to Train an RNN

Long Term Short Term Memory (LSTMs)

**7.1 Modeling Sequences: a brief overview**

In this section we'll overview of various types of models that have been used for sequences.

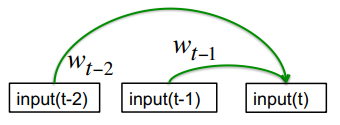
* We'll start with the simplest kinds of model, which is Auto Regressive Model, they just *try and predict* the ***next term*** in a sequence from ***previous terms***.
* We'll discuss about *more elaborate variants of them* using *hidden units*.
* Then we'll talk about, more interesting kinds of models, that have ***hidden state***, and ***hidden dynamics***. These include
* linear dynamical systems and
* hidden Markov models.
* Most of these are quite complicated kinds of models, so we'll not dive into the details of them. The main point of mentioning them is to be able to show how *RNNs are related to models of that kind*.
* **Getting targets when modeling sequences**

When we're using ***Machine Learning*** to model ***sequences***, we often want to turn *one sequence into another sequence*. For example, we might want to

* Translation: Turn English words into French words
* Speech recognition: Take a ***sequence of sound pressures*** and turn it into a ***sequence of word identities*** which is what's happening in speech recognition.
* Sometimes we don't have a ***separate target sequence***, and in that case we can get a teaching signal by trying to predict the next term in the input sequence.
* So the ***target output sequence*** is simply the ***input sequence*** with an ***advance*** of one time-step.
* This seems much more ***natural*** than trying to ***predict one pixel*** in an image from the all other pixels, or ***one patch*** of an image from the rest of the image.
* One reason it probably seems more natural is that: for ***temporal sequences***, there is a ***natural order*** for the predictions. Whereas for images it's not clear what you should predict from what. But in fact a similar approach works very well for images.
* ***Supervised or Unsupervised?*** When we predict the *next term in a sequence*, it *blurs the distinction, between supervised and unsupervised learning*.
* So we use methods that were *designed for supervised learning* to ***predict*** the ***next term***. But we *don't require separate* ***teaching signal***. So in that sense, it's *unsupervised*.
* **Memoryless models for sequences**

Let's do a quick review of, before we get on to using ***Recurrent Neural Nets*** to model ***Sequences***.

* Autoregressive models: Predict the next term in a sequence from a fixed number of previous terms using “delay taps”.



* ***Auto regressive model*** is a nice simple model for sequences that ***doesn't have*** any ***memory***.
* It simply takes some previous terms in the sequence and try and predict the next term, basically as a ***weighted average*** of ***previous terms***.
* The ***previous terms*** might be ***individual values*** or they might be ***whole vectors***. A ***Linear Auto Regressive Model*** would just take a *weighted average* of those to *predict* the *next term*.

|  |  |
| --- | --- |
| * Feed-forward neural nets: These generalize autoregressive models by using one or more layers of non-linear hidden units. e.g. Bengio’s first language model. * We can make ***autoregressive models*** considerably more ***complicated*** by adding ***hidden units***. * In a ***Feed-Forward Neural Net***, we might take some ***previous*** *input terms*, put them through some ***hidden*** *units*, and *predict* the ***next*** *term*. |  |

* **Beyond memoryless models**

Memory-less models are only one ***subclass of models*** that can be used for ***sequences***. We can think about ways of ***generating sequences***, and one very *natural way to generate a sequence* is to have a model that has some ***hidden state*** which has its ***own internal*** ***dynamics***.

* The hidden state ***evolves*** according to its ***internal dynamics***, and the hidden state also ***produces observations***, and we get to see those observations. That's a much more interesting kind of model.
* I.e. if we give our generative model some hidden state, and if we give this hidden state its own internal dynamics, we get a much more interesting kind of model.
* It can ***store information*** in its ***hidden state*** for a long time. Unlike the memoryless models, *there's no simple bound*, to how far we have to ***look back*** before we can be sure it's ***not affecting*** things.
* If the ***dynamics*** of the ***hidden state*** is ***noisy*** and the way it generates ***outputs*** from its hidden state is ***noisy***, then by observing the output of a generative model like this, we can ***never know*** its ***exact hidden state***.
* The best you can do is to infer ***probability distribution*** over the *space of all possible hidden state vectors*. You can know that it's probably in *some part of the space* and not *another part* of the space, but you ***can't pin it down*** exactly.
* This inference is only ***tractable*** for two types of ***hidden state model***.
* With a generative model like this, if you *get to observe what it produces*, and you now try to ***infer*** what the ***hidden state was***, in general that's very ***hard***, but there're *two types of hidden state model* for which the ***computation*** is ***tractable***.
* i.e., there's a fairly *straightforward computation* that allows you to ***infer*** the *probability distribution* over the ***hidden state vectors*** that *might* have *caused* the ***data***.
* Of course when we do this and apply it to ***real data***. We're assuming that the *real data* is *generated* by our *model*. So that's typically what we do when we're *modeling things*. We assume the *data was generated by the model* and then we ***infer*** what *state* the *model* must have been in, *in order to generate that data*.
* The next three topics (sub-sections) are mainly intended for people who already know about the two types of hidden state model we're about to describe. The point is that we make it clear how ***recurrent neural networks (RNNs)*** differ from those ***standard models***.
* If you can't follow the details of the two standard models, don't worry too much. That's not the main point.
* **Linear Dynamical Systems (engineers love them!)**
* One standard model is a linear dynamical system. It's very widely used in engineering. This is a ***generative model*** that has ***real valued hidden state*** that cannot be observed directly.

|  |  |
| --- | --- |
| * The ***hidden state*** has ***linear dynamics***, shown by those red arrows on the right. The dynamics has ***Gaussian noise***, so that the ***hidden*** ***state evolves probabilistically***. The hidden state ***produces*** the ***observations*** using a linear model with ***Gaussian noise***. * ***Driving inputs:*** There may also be ***driving inputs***, shown at the ***bottom layer***, which directly influence the hidden state. So the *inputs, influence the hidden state directly*, the ***hidden state*** determines the ***output***. |  |

|  |  |
| --- | --- |
| * To predict the next ***output*** of a system like this, we need to be able to ***infer*** its ***hidden state***. And these kinds of systems are used, for example, for ***tracking missiles***. In fact, one of the earliest uses of ***Gaussian distributions*** was for trying to ***track planets*** *from noisy observations*. * ***A linearly transformed Gaussian is a Gaussian:*** One nice property that a Gaussian has is that if you ***linearly transform a Gaussian*** you get *another Gaussian*. Because *all the* ***noise*** *in a* ***linear dynamic system*** is Gaussian. It turns out that the *distribution over the hidden state* given the *observation (data)* so far (i.e. given the output so far), is also a Gaussian. * ***Kalman filtering:*** It's a ***full covariance Gaussian***, and it's quite complicated to compute what it is. But it can be *computed efficiently*. And there's a technique called Kalman Filtering. * This is an efficient *recursive way of updating* your ***representation*** *of the* ***hidden state*** *given a* ***new observation***. |  |

* So, to summarize, *Given observations of the output of the system*,
* We can't be sure what hidden state it was in, but
* We can, estimate a *Gaussian distribution over the possible hidden states* it might have been in. Always assuming, of course, that our model is a *correct model of a reality we're observing*.
* **Hidden Markov Models (HMMs: computer scientists love them!)**

A hidden Markov model uses ***Discrete Distributions*** rather than ***Gaussian distributions***. So it is a *different kind of hidden state model*. Since it's based on ***discrete mathematics***, computer scientists love these ones.

* Hidden Markov Models (HMM) have a *discrete one-of-N hidden state*.
* In a hidden Markov model, the hidden state consists of a ***one-of-N Choice***. So there're a number of things called states. And the system is always in exactly *one of those states*.
* The ***transitions*** between *states* are ***probabilistic***.
* Transitions between states are ***stochastic*** and they're controlled by a transition matrix which is simply a *bunch of* ***probabilities*** that say, *"If you're in state one at time , What's the probability of you going to state three at time ?"*
* The output model is also stochastic.
* The ***outputs*** produced by a ***state*** are ***stochastic***. So, the ***state*** *that the system is* ***in***, *doesn't completely determine* *what output it produces*. There's some ***variation*** in the ***output*** that each state can ***produce***.
* Because of that, we *can't be sure* which ***state*** produced a given ***output***. So the state is “hidden”.
* In essence, the *states* are *hidden behind this probabilistic veil*, and that's why they're called hidden. We cannot be sure which state produced a given output.
* Historically the reason ***hidden units*** in a *neural network* are called ***hidden***, is because I (Geoffrey Hinton) like this term. It sounded *mysterious*, so I *stole* it from *neural networks*.

|  |  |
| --- | --- |
| * It is easy to *represent* a *probability distribution* across **N states** with **N numbers**. * The nice thing about a *hidden Markov model* is, we can ***represent*** the ***probability distribution*** across its ***discreet states***. * So, even though we don't know *"what state it's in"* for sure, we can easily represent the ***probability distribution***. * To predict the next output we need to infer the probability distribution over hidden states. * To ***predict*** the ***next*** ***output*** from a *Hidden Markov Model*, we need to *infer what hidden state it's probably in*. And so we need to get our hands on that *probability distribution*. * It turns out there's an easy method based on dynamic programming that allows us to take *the observations we've made* and from those compute the *probability distribution across the hidden states*. |  |

* ***HMMs have efficient algorithms for inference and learning:*** Once we have that distribution, there's a *nice elegant learning algorithm* hidden Markov models, and that's what made them so appropriate for ***speech***. And in the 1970s, they took over ***speech recognition***.
* **A fundamental limitation of HMMs**

There's a fundamental limitation of HMMs. It's easiest to understand this limitation, if we consider what happens when *a hidden Markov model* ***generates data***. At each time step when it's generating, it selects one of its hidden states.

* So if it's got **N** ***hidden states***, the ***total information*** stored in the ***hidden state*** is at most ***bits*** and that's all it knows about *what it's done so far*.
* So with ***N hidden states*** it can only *remember* ***log(N)*** *bits* about what it generated so far.
* Consider the information that the ***first half*** of an utterance ***contains*** about the ***second half***: Let's consider *how much information* a ***hidden Markov model*** can convey to the ***second half of an utterance*** it produces from the ***first half***.
* ***Syntax needs to fit*** *(e.g. number and tense agreement):* So imagine it's already produced the *first half of an utterance*. And now it's going to have to produce the *second half*. And remember, its memory of what it said for *the first half is in* which of the n-states it's in. So its memory only has ***bits*** of information in it. To produce the *second half* that's ***compatible*** with the *first half*, we must make the syntax fit.
* So for example, the number and tense must agree.
* ***Semantics needs to fit:*** It also needs to make the semantics fit. It can't have the *second half of the sentence* be about something *totally different* from the *first half*.
* ***Intonation needs to fit:*** Also the intonation needs to fit so it would look very silly if the, *intonation contour* completely changed *halfway* through the sentence.
* *The* ***accent****,* ***rate****,* ***volume****, and* ***vocal tract******characteristics*** *must all fit:* There's a lot of other things that also have to fit. The accent of the *speaker*, The rate they're *speaking* at, how loudly they're *speaking (volume)* and the vocal tract characteristics of the speaker. All of those things must *fit* between the *second half* of the sentence and the *first half*.
* And so if you wanted a hidden Markov model to actually *generate a sentence*, the ***hidden state*** has to be *able to convey all that information* from the ***first half*** to the ***second half***.
* Now the **problem** is that all of those aspects combined could easily come to a **100 bits of information**. So the ***first half*** of the sentence (utterance) needs to *convey* a **100 bits** of information to the *second half* and that means that the hidden Markov model needs states and that's just too many.
* **Recurrent neural networks (RNN)**

Those limitations of HMMs bring us to ***Recurrent Neural Networks (RNNs)***. They have a much more efficient way of remembering information.

|  |  |
| --- | --- |
| * They're very powerful because they combine two properties:  1. Distributed hidden state allows them to store a lot of information about the past efficiently.  * They have distributed hidden state. That means, several different units can be active at once. So they can remember several different things at once. They don't just have one active unit.  1. Non-linear dynamics allows them to update their hidden state in complicated ways.  * They're also nonlinear. We see, a ***linear dynamical system*** has a ***whole hidden state vector***. So it's got more than one value at a time, but those values are ***constrained*** *to act in a* ***linear way*** so as to make ***inference*** easy, and in a ***RNN*** we allow the dynamics to be much more complicated. |  |

With *enough* ***neurons*** and *enough* ***time***, a ***RNN*** can compute anything that can be computed by your computer. It's a very powerful device.

* **Do generative models need to be stochastic?**
* Linear Dynamical Systems and HMMs are both stochastic models. That is the ***dynamics*** *and the* ***production of observations*** *from the* ***underlying state***, both involve intrinsic noise.
* The question is: *"Do models need to be like that?"*. Well one thing to notice is that the ***posterior probability distribution*** over their ***hidden*** ***states*** in either a ***Linear Dynamical System*** or ***HMM*** is a *deterministic function* of the data that you've seen so far.
* i.e. the posterior probability distribution over their hidden states given the *observed data so far* is a *deterministic function of the data*.
* That is the inference algorithm for these systems ends up with a probability distribution, and that *probability distribution* is just a *bunch* *of numbers*, and those numbers are a *deterministic function of the data* so far.
* ***Recurrent neural networks are deterministic:*** In a ***Recurrent Neural Network (RNN),*** you get a bunch of numbers that are a *deterministic function of the data* so far.
* So think of those *numbers* that *constitute the hidden state* of a RNN are very like the *probability distribution* for these simpler stochastic models (***Linear Dynamical System*** or ***HMM***).
* i.e. the *hidden state of an RNN* as the equivalent of the *deterministic probability distribution over hidden states* in a Linear Dynamical System or Hidden Markov Model.
* **Recurrent neural networks**

What kinds of behavior can RNNs exhibit?

* ***They can oscillate:*** That's obviously good for things like ***motor control***, where when you're ***walking***, for example, you want to know ***regular oscillation***, which is your ***stride***.
* ***They can settle to point attractors:*** That might be good for ***retrieving memories***. Later we'll discuss the ***Hopfield nets*** where they use the ***settling*** to ***point attractors*** to ***store memories***.
* So the idea is: you have a sort of *rough idea* of what you're *trying to retrieve*. You then let the system *settle down to a stable point* and those ***stable points*** correspond to the ***things you know about***. And so by *settling to that stable point* you *retrieve a memory*.
* ***They can behave chaotically:*** If you set the ***weights*** in the ***appropriate regime***. Often, chaotic behavior is ***bad for information*** ***processing***, because in information processing, you want to be able to *behave reliably*. You want to achieve something. There are some circumstances where ***it's a good idea***.
* If you're up against a *much smarter* ***adversary***, you probably can't outwit them, so it might be a ***good idea*** *just to* ***behave randomly***. And one way to get the *appearance of randomness* is to *behave chaotically*.
* One nice thing about RNN's is that it could learn to ***implement*** lots of ***little programs***, using different subsets of its hidden state.
* And each of these *little programs* could *capture a nugget of knowledge*. And all of these things could run in parallel, and interact with each other in *complicated ways* (interacting to produce very complicated effects).
* Unfortunately the *Computational Power of RNNs* makes them very ***hard to train***. For many years, we couldn't exploit the computational power of RNNs.
* There was some heroic efforts. For example, Tony Robinson managed to make quite a good speech recognizer using ***Recurrent Nets***. He had to do a lot of work *implementing* them on a *parallel computer* built out of TRANSPUTERS.
* And it was only recently that people managed to produce RNNs that outperformed Tony Robinson's.

**7.2 Training RNNs with Back Propagation**

In this section we're going to talk about the ***Back Propagation through Time*** algorithm, it's the standard way to train the RNN.

* The algorithm is really quite simple once you have seen the equivalents between a ***Recurrent Neural Network (RNN)*** and a ***Feed Forward*** ***Neural Network*** that has ***one layer*** *for* ***each time step***.
* We'll also talk about ways of *providing* ***input***, and *desired* ***outputs***, to RNNs.

|  |  |
| --- | --- |
| * **The equivalence between *Feed-Forward* nets and *Recurrent Nets***   The diagram shows a simple Recurrent Net with *3 interconnected neurons*. We're going to assume there's a time delay of 1 in using each of those ***connections*** and that the network *runs in* ***discrete time***, so the clock that has integer ticks.   * The key to understanding how to train a recurrent network is: to see that a ***Recurrent Network*** is really just the same as a ***Feed Forward*** ***Network***, where you've ***expanded*** *the recurrent* ***network*** *in* ***time***. |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| * The recurrent network starts off in some ***initial state***. Shown at the bottom layer, time=0. * And then uses the ***weights*** of its connections to *get a new state*, shown at time=1. * Then it uses the ***same weights*** again to get *another new state*, at time=2 * And it uses the ***same weights*** again to get *another new state* at time=3 and so on.   So it's really just a ***Layered Feed Forward Network***, where the ***weight*** is a *constraint to be the* ***same*** at ***every layer*** i.e. the *Recurrent Net* is just a *Layered Net* that *keeps* ***reusing*** *the* ***same weights***. |  |

* **Reminder: Backpropagation with weight constraints**

It is easy to *modify the backprop algorithm* to *incorporate* ***linear constraints*** between the ***weights***.

* ***Backprop*** is *good at learning* when there are weight constraints. We saw this for Convolutional Nets and we can actually ***incorporate*** any ***linear constraint*** quite easily in ***backprop***.

|  |  |  |
| --- | --- | --- |
| * So we ***compute the gradients*** *as usual, as if the* ***weights were not constrained***. And then we ***modify the gradients***, so that they ***satisfy the constraints***. * So if we want , (we start off with an equal) and then we need to make sure that the is equal to the , i.e. . * And we do that by simply taking the derivative of the error w.r to and and adding or averaging them, and then applying the ***same*** ***quantity*** for ***updating*** both and . |  |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |

* So if the ***weights*** *started off satisfying* the constraints, they'll *continue to satisfy the constraints*.
* **Backpropagation through time**

The ***Backpropagation Through Time Algorithm*** is just the name for what happens when you think of a Recurrent Net as a ***layered feed forward net*** with shared weights, and you train it with ***Backpropagation*** (i.e. train the feed-forward net with weight constraints).

* So, we can think of this training algorithm in the ***time domain***.
* The ***forward pass*** builds up a ***stack of activities*** (of all the units) at *each time slice* (time step).
* 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.
* After the backward pass we can *add together the derivatives* at all the *different time step* for each *particular weight*. And then ***change*** *all the copies of that* ***weight*** by the *same amount* which is *proportional to the sum or average of all those derivatives*.
* **An irritating extra issue**

There is an ***irritating extra issue***. If we *don't specify the* ***initial state*** *of the* ***all the units***, for example, if some of them are ***hidden*** or ***output*** units, then we have to *start them off in some particular state*.

* ***We need to specify the initial activity state of all the hidden and output units:***
* We could just ***fix*** those initial ***states*** to have some ***default value*** like **0.5**, but that might make the system work not quite as well as it would otherwise work if it had some *more sensible* ***initial value***.
* ***But it is better to treat the initial states as learned parameters:*** We learn them in the same way as we learn the weights.
* We can actually ***learn*** the ***initial states***. We treat them like parameters rather than activities and we ***learn*** *them the* ***same way*** *as learned the* ***weights***.
* We start off with an **initial random guess** for the ***initial states***. That is the *initial states* of *all the* ***units*** *that* ***aren't input units***.
* Then at the *end of each training sequence* we ***back-propagate through time*** all the way ***back*** to the ***initial states***.
* And that gives us the ***gradient*** of the ***Error Function*** w.r.to the each ***initial state***.
* ***Adjust the initial states by following the negative gradient:*** We then just, *adjust the initial states* by following that *gradient*. We go *downhill* in the *gradient*, and that gives us *new initial states* that are *slightly different*.
* **Providing input to recurrent networks**

|  |  |
| --- | --- |
| There's many ways in which we can *provide* the input to a ***Recurrent Neural Net***. |  |
| * ***Specify the initial states of all the units:*** We could ***specify*** the ***initial state*** of all the units. That's the most natural thing to do when we think of a ***Recurrent Net***, like a ***Feed Forward Net*** *with constrained weights*. |  |
| * ***Specify the initial states of a subset of the units:*** We also could *specify* the *initial state* of just a ***subset*** *of the* ***units***. |  |
| * ***Specify the states of the same subset of the units at every time step*** *(natural way to model most sequential data):* Or, we can *specify* the *states at every time step* of the *subset of the units* and that's probably the most *natural way to input* ***sequential*** ***data***. |  |

* **Teaching signals for recurrent networks**

|  |  |
| --- | --- |
| Similarly, there's many way we can specify targets for a ***Recurrent Network***. |  |
| * *Specify desired* ***final activities*** *of* ***all the units****:* When we think of it as ***Feed Forward Network*** *with constrained weights*, the natural thing to do is to specify the desired final states for all of the units. |  |
| * *Specify desired* ***activities of all units*** *for the* ***last few steps*** *(good for learning attractors):* If we're trying to train it to ***settle*** *to some* ***attractor***, we might want to ***specify*** *the* ***desired states*** not just for the ***final time steps*** but for ***several time steps***. That will cause it to actually *settle down there*, rather than *passing through some state and going off somewhere else*. * So by specifying ***several states*** *of the* ***end***, we can force it to learn ***attractors***. * *It is easy to* ***add in extra error derivatives*** *as we* ***backpropagate****:* It's quite easy as we *back-propagate* to *add-in derivatives* *(extra error derivatives)* that we get from *each time step*. So the back-propegation *starts at the* ***top***, with the ***derivatives*** for the ***final time step***. * And then as we *go back through* the *layer before the top* we add in the derivatives for that layer, and so on. * So it's really very little extra effort *to have* ***derivatives*** *at many* ***different layers***. |  |
| * *Specify the desired* ***activity of a subset of the units*** *(other units are input or hidden units):* We could ***specify*** *the desired* ***activity*** *of a* ***subset of units***which we might *think* of as ***output units***. * And that's a very natural way to ***train*** a *Recurrent Neural Network* that is meant to be *providing a continuous output*. |  |

**7.3 A toy example of Training an RNN**

In this section we'll discuss how a RNN solves a toy problem. This problem demonstrates what it is we can do with RNNs that *cannot be done conveniently* with ***feed forward neural networks***.

* The problem is ***adding up*** *two* ***binary numbers***.
* After the ***RNN*** has learned to solve the problem, it's interesting to look at its *hidden states*, and see how they relate to the *hidden states in a* ***finite state automaton*** that's solving the same problem.
* **A good toy problem for a recurrent network**

|  |  |
| --- | --- |
| So we consider the problem of *adding up two binary numbers*. We could train a *Feed-Forward Neural Network* to do that. And the diagram on the right shows a network that gets some *inputs* and produces some *outputs*.   * But there are ***problems*** with using a *Feed-Forward Neural Network*. There are obvious regularities that it cannot capture efficiently. * We must *decide in advance*, what the *maximum number of digits* is (in each number), for both of the input numbers and for the output number. |  |

* And more importantly, the ***processing*** that we apply to the *different bits* of the ***input numbers***, doesn't generalize. That is, when we *learn how to* ***add up*** *the* ***last two digits*** and deal with the ***carries***, that knowledge's in some weights. And as we go to a *different part* of a *long* *binary number*, the *knowledge* will have to be *in different weights*. ***So we won't get automatic generalization***.
* The *processing* applied to the *beginning of a long number* **does not generalize** to the *end of the long number* because it uses different weights.
* As a result, although you can train a neuron feed-foward neural network, and it will eventually *learn* to do *binary addition* on *fixed-length* *numbers*, it's not an elegant way to solve the problem.
* i.e., feed-forward nets ***do not generalize*** well on the binary addition task.
* **The algorithm for binary addition**

Following is a picture of the ***algorithm*** for ***binary addition***.

|  |  |
| --- | --- |
| This is a ***Finite State Automaton***. It decides what *transition* to make by *looking at the next* *column*. It prints after making the transition. It *moves from right to left* over the two input numbers.   * The states shown here are like the ***states*** *in a* ***hidden Markov model***, except they're not really hidden. * The system is in *one state at a time*. |  |

* When it ***enters a state*** it performs an ***action***.
* So it either prints 1 or 0 and when *it's in a state* it ***gets*** some ***input***, which is the two numbers in the next column.
* And that ***input*** causes it to *go into a new state*.
* So if you look on the ***top right***, it's in the *carry state* and it's just printed a 1.
* If it sees a (1, 1), it goes back in to the same state and print another 1.
* If however it sees a (1, 1) or (0, 1), it goes into the *carry state* but prints a 0.
* If it sees a (0, 0), it goes into the *no carry state*, and prints a 1. And so on.

|  |  |
| --- | --- |
| * **A recurrent net for binary addition**   A ***recurrent neural net*** for *binary addition* needs to has ***two*** *input units* and ***one*** *output unit*.   * It's given ***two input digits*** at each ***time step***. * It also has to produce an ***output*** at each ***time step***. |  |

* ***The desired output at each time step is:*** The *output* for the *column* that was provided as input *two time steps ago*. The reason we need a delay of two time steps, is that it takes
* It takes *one time step to* ***update*** *the* ***hidden units*** based on the inputs (two input digits), and
* *another time step to* ***produce*** *the* ***output*** *from the hidden state*. i.e.it takes another time step for the hidden units to cause the output.
* **The connectivity of the network**

***So the net looks like this:*** we only gave it ***3 hidden units*** (sufficient to do the job). It would *learn faster* with *more hidden units*, but it can do it with 3.

|  |  |
| --- | --- |
| * *The* ***3 hidden units*** *are* ***fully interconnected*** *in both directions:* The 3 hidden units are fully interconnected and they have connections in both directions that don't necessarily have the same weight. In fact in general they don't have the same weight. * The ***connections*** between ***hidden units*** allow the *pattern of one time step* to insensate the *pattern of the next time step*. * This allows a *hidden activity pattern at one time step* to vote for the *hidden activity pattern at the next time step*. |  |

* The *input units* have *feed forward connections* to the ***hidden units*** and that's how it sees the two digits in a column. That allows to *vote* for the *next hidden activity pattern*.
* And similarly, the *hidden units* have *feed forward connections* to the ***output unit*** and that's how it produces its *output*.
* What the network learns: It's interesting to look at what the ***RNN*** learns. It learns *four distinct patterns* of activity in its ***3 hidden units***.
* And these patterns correspond to the nodes in the ***finite state automaton*** for binary addition.
* We must not confuse the ***units*** in a ***neural network***, with the nodes in a finite state automaton. The nodes in the ***finite state automaton*** correspond to the ***activity vectors*** of the ***recurrent neural network***. I.e. nodes are like activity vectors.
* The automaton is *restricted* to being *exactly* ***one state*** *at each time*. And similarly, the hidden units are *restricted* to have *exactly* ***one******activity vector*** *at each time* in the **Recurrent Neural Network**.
* So a ***Recurrent Neural Network*** can emulate a ***Finite State Automaton*** but it's *exponentially more powerful* in its representation. With *hidden neurons*, it has possible *binary activity vectors* (but only weights).
* Of course it only has weights so it can't necessarily make full use of all that representational power. But if the ***bottleneck*** is in the ***representation***, a ***RNN*** can do much better than a ***Finite State Automaton***.
* This is important when the ***input stream*** has *two separate things going on at* ***once***.
* A ***Finite State Automaton*** needs to *square* its *number of states* in order to deal with the fact that there are *two things going on at once*.
* A RNN only needs to ***double*** its ***number of hidden units***. By doubling the number of units, it does of course *square* the *number of binary* *vector states* that it has.

**7.4 Why it is Difficult to Train an RNN**

|  |  |
| --- | --- |
| In this section we'll talk about the ***exploding*** and ***vanishing*** *gradients problem*, which makes it difficult to train RNN. For many years, researchers in Neural Networks thought they would *never be able to train these networks* to model *dependencies over long time-periods*.   * We'll also discuss *four different ways* to train those networks. * **The backward pass is linear**   To understand why it's so difficult to train RNNs, we have to understand a very important difference between the forward and backward passes in a RNN.   * There is a big difference between the forward and backward passes: * Forward: In the forward pass, we used squashing functions, like the ***logistic***, to *prevent* the *activity vectors* from ***exploding***. * If you look at the above picture on the right, *each neuron* is using a logistic unit shown by that **blue curve** and it can't output any value greater than **1** or less than **0**, so, that stops explosions. |  |

* Backward: The backward pass, however, is ***completely linear***. Most people find this very surprising. If you *double* the *error derivatives* at the *final layers* of this net, ***all the*** error derivatives ***will double*** when you ***back propagate***.
* If you look at the **red dots** that we put on the **blue curves**, we'll suppose those are the **activity levels** of the neurons on the *forward pass*.
* When you *back propagate*, you're using the ***gradients*** of the ***blue curves*** at those ***red dots***. So the ***red lines*** are meant to draw the ***tangents*** *to the* ***blue curves*** *at the* ***red dots***.
* Once you *finish the forward pass*, the slope of that tangent is fixed. You then *back propagate* and the *back propagation* is like *going* ***forwards*** *though a* ***linear system*** in which the ***slope*** *of the* ***non-linearity*** has been ***fixed***.
* Of course, ***each time*** you back propagate, the ***slopes*** will be ***different*** because they were ***determined*** *by the* ***forward pass***.
* But *during the back propagation*, it's a ***linear system*** and so it suffers from a *problem of linear systems*, which is: when you ***iterate***, they tend to *either* ***explode*** *or* ***die***.
* So the *forward pass determines the slope* of the *linear function* used for *backpropagating* through *each neuron*.
* **The problem of exploding or vanishing gradients**
* What happens to the ***magnitude of the gradients*** as we ***backpropagate*** through many layers?
* *If the weights are small, the gradients shrink exponentially:* When we back-propagate through many layers, if the *weights are small* the gradients will shrink and become *exponentially small*. And that means that: when you back-propagate through time, ***gradients*** that are *many steps earlier than the* ***targets*** arrive, will be *tiny*.
* If the weights are big the gradients grow exponentially: Similarly, if the *weights are big*, the gradients will explode. And *that means: when you back-propagate through time, the* ***gradients*** will get *huge* and *wipe out all your* ***knowledge***.
* In a *feed-forward neural net*, unless it's very deep, these problems aren't nearly as bad because we typically only have a *few* *hidden layers*.
* So typical feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.
* But as soon as we have a RNN trained on a long sequence, for example 100 time steps, then the gradients can easily explode or vanish.
* 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* and,
* So, they'll either explode or vanish.
* We can***avoid this*** by *initializing the weights* very carefully. (more recent work, shows that indeed careful initialization of the weights does make things work much better).
* But even with *good initial weights*, it's *hard to* ***detect*** *the* ***dependency*** of the *current* ***target******output*** on an *input from many time-steps* *ago*.
* So it's hard to make the output depend on things that happened a long time ago.
* So RNNs have *difficulty dealing with* ***long-range dependencies***.
* **Why the back-propagated gradient blows up**

Following is an example of *exploding* and *dying gradients* for a ***system*** that's trying to ***learn attractors***. Suppose we're trying to train a RNN, so that whatever state we started in, it *ends up* in one of these two *attractor states*.

|  |  |
| --- | --- |
| * So we're gonna learn a *blue basin of attraction* and a *pink basin of attraction*. * *If we start a trajectory within an attractor, small changes in where we start make no difference to where we end up (****vanishing gradients****):* If we *start anywhere* within the ***blue basin of attraction***, we will ***end up*** at the ***same point***. * What that means is that, *small differences in our initial state* ***make no difference*** to where we ***end up***. |  |

* So the *derivative* of the *final state* w.r. to *changes in the initial state*, is 0. That's vanishing gradients. When we ***back propagate*** through the dynamics of this system we will discover: *there's no gradients from where you start*, and the *same* with the *pink basin of attraction*.
* *Start almost exactly on the boundary, tiny changes can make a huge difference (****exploding gradient****):* If however, we start very close to the boundary between the two attractors. Then, a tiny difference in where we start, that's the other side of the watershed, makes the huge difference to where we end up, that's the exploding gradient problem.

And so whenever you're trying to use a RNN to learn attractors like this, you're bound to get vanishing or exploding gradients.

* **Four effective ways to learn an RNN**

There are at least four effective ways to learn a RNN.

* Long Short Term Memory (LSTM): The first is a method called long short term memory and I'll talk about that more in this lecture.
* ***The idea is:*** we actually change the architecture of the neural network to make it good at remembering things.
* We'll make the RNN out of little modules that are designed to remember values for a long time.
* Hessian Free Optimization: This method deal with the vanishing gradients problem by using a much better optimizer that can detect directions with a tiny gradient but even smaller curvature. We'll talk about that in the next chapter.
* The real problem in optimization is to detect small gradients that have an even smaller curvature. Heissan-free Optimization, tailored to Neural Nets is good at doing that.
* Echo State Networks: This method really kind of evades (avoid) the problem. In this method, we *initialize* the input-hidden and hidden-hidden and output-hidden *connections very carefully* so that the ***hidden state*** has a ***huge reservoir*** of *weakly coupled oscillators* which can be selectively driven by the input. ***ESNs*** *only need to learn* the ***hidden-output connections***.
* ***What we do is:*** we carefully initialize the ***input to hidden*** *weights* and we very carefully initialize the ***hidden to hidden*** *weights*, and also *feedback weights* from the ***outputs to the hidden*** units.
* And the idea of this careful initialization is to make sure that the ***hidden state*** has a huge *reservoir* of *weakly coupled oscillators*.
* So if you hit it with an input sequence, it will ***reverberate*** *for a long time* and those ***reverberations*** *are* ***remembering*** *what happened in the input sequence*.
* You then try en-couple those reverberations to the output you want.
  + So the only thing that learns in an ***Echo State Network*** is the *connections between the* ***hidden units*** *and the* ***outputs***.
* And if the ***output units*** are ***linear***, that's very easy to train. So this hasn't really learned the recurrent-bit. It's used a fixed random recurrent bit, but a carefully chosen one and then just learned the *hidden to output* connections.
* Good initialization with momentum: In this method we use ***momentum*** with the kind of ***initialization*** that was being used for *Echo State Networks (ESN)* and that makes them work even better.
* Initialize like in Echo State Networks, but then learn all of the connections using momentum.

So it was very clever to find out how to initialize these ***Recurrent Networks*** so that they'll have *interesting dynamics*, but they work even better if you now *modify that dynamic slightly* in that direction that will help with the task at hand.

**7.5 Long Term Short Term Memory**

In this section, we'll discuss the ***LSTM*** (Long Short Term Memory), which is the most popular approach to train a ***RNN*** (recurrent neural networks).

* In LSTM approach, we consider the ***dynamic state*** *of a neural network* to be a *short term memory*.
* ***The idea is:*** you want to make that short term memory *last for a long time*.
* This is done by creating *special modules* that are designed to *allow* ***information*** *to be* ***gated in***, and then information to be ***gated out*** when ***needed***.
* And in the *intermediate period*, the ***gate*** is ***closed***, so the stuff that arrives in the intermediate period ***doesn't interfere*** *with the* ***remembered state***.

LSTM has been very successful for tasks like *recognizing handwriting*, where it's won a number of competitions.

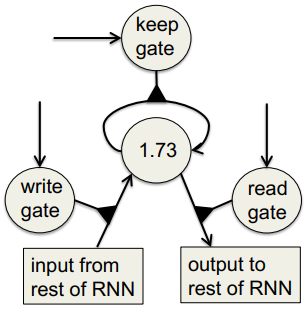
* **Long Short Term Memory (LSTM)**

In 1997, Hochreiter & Schmidhuber published a paper in neural computation that solved the problem of getting a RNN to *remember things* *for a long time*. Their recurrent nets could remember things for hundreds of time steps.

* They did this by designing a memory cell that used logistic and linear units with *multiplicative interactions*.
* So information gets into the memory cell, whenever a ***logistic "write" gate*** is turned on.
* The *rest of the Recurrent Network* determines the ***state*** *of that* ***write gate***, and when the *rest* of the *recurrent network* wants *information* *to be stored*, it turns the ***write gate*** ***on***, and whatever the *current input* from the *rest of the net* to the *memory cell* is, gets ***stored*** in the ***memory*** cell.
* The *information* stays in the *memory* cell so long as its ***"keep" gate*** is on. Again, the *rest of the system* is determining the ***state*** of a ***logistic "keep" gate***, and if it keeps it ***on***, then the *information* will *stay there*.
* Finally, the *information* gets *read* from the *memory cell* so that it then *goes off to the rest of the* ***RNN*** and *influences future states* and it's read by turning on a ***"read" gate***. Which again is a ***logistic unit*** controlled by the rest of the neural network.
* **Implementing a memory cell in a neural network**

The ***memory cell*** actually stores an ***analog value***, so we can think of it as a ***linear neuron*** *that has an* ***analog value*** and *keeps writing that value to itself* at ***each time step*** by a *weight of* ***1***, so the information just stays there.

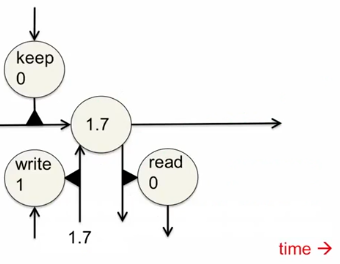
|  |  |
| --- | --- |
| * To *preserve information for a long time* in the activities of an RNN, we use a ***circuit*** that implements an ***analog memory cell***. * A ***linear unit*** *(neuron)* that has a ***self-link*** with a ***weight of 1*** will maintain its state. * The weight of 1 is determined by a ***"keep gate"*** so the rest of the system determines the state of that ***logistic keep gate*** and if it puts it into a state of **1** or close to **1** the *information* just *cycles around* and that value of 1.73 will *stay there*. * As soon as the rest of the system wants to *get rid of that value*, all it has to do is *set the* ***keep gate*** to have *a value of 0* and the *information* will *disappear*. |  |
| * *Information is* ***stored*** *in the cell by* ***activating*** *its* ***write gate****:* To *store the information in the memory cell*, the *rest of the system* has to **turn** **on** the ***"write gate"***. * And then *whatever input* is being provided to the memory cell from the rest of the system *will get* ***written*** into the *memory cell*. * *Information is* ***retrieved*** *by* ***activating*** *the* ***read gate****:* To read the information from the memory cell, the rest of the system **turns on** the logistic ***"read gate"*** and then, the *value* in the memory cell *comes out and affects the rest of the RNN*. * The point of using ***logistic units*** is that we can ***back-propagate*** through them because they have ***nice derivatives***, and that means we can learn to use this kind of circuit over *many time steps*. |  |



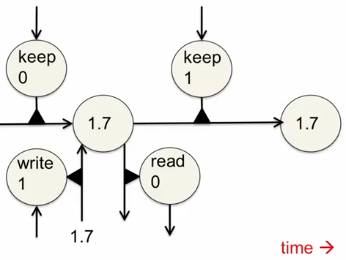
* **Backpropagation through a memory cell**

Let's discuss the picture of what *Backpropagation through a memory cell* looks like.

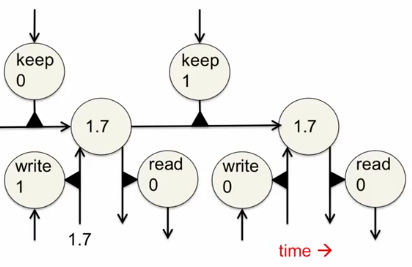
* Forward pass: First we're going to do a forward pass. So at the initial time, let's suppose that:
* The keep gate was set to 0, so we wiped out whatever information was in the memory cell before,
* And the write gate is set to 1. So the value of **1.7** that is coming from the *rest of the recurrent neural network* gets ***written*** into the ***memory*** cell.
* Also we're *not going to read* it at this time, so the read gate is set to 0.



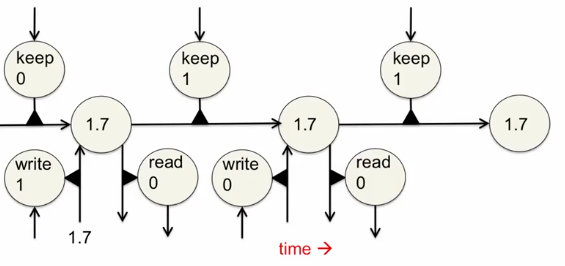
* At the ***next time step***, we set the keep gate to 1, or rather the rest of the, neural network has to set the keep gate to 1. It means that the value **1.7** is *written* into the *memory* cell, i.e. it's stored.



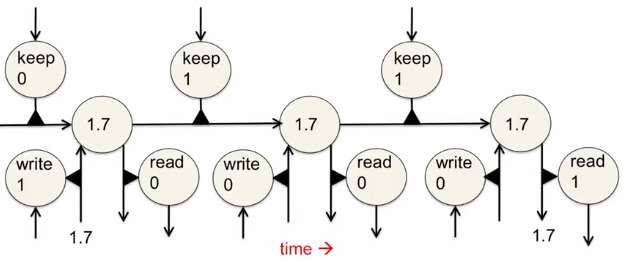
* Then, we're going to set the write gate to 0 and the read gate to 0, so the information *isn't influenced by* what's going on in the *rest of the net*, and it *doesn't influence* what's going on in the rest of the net. It's insulated.



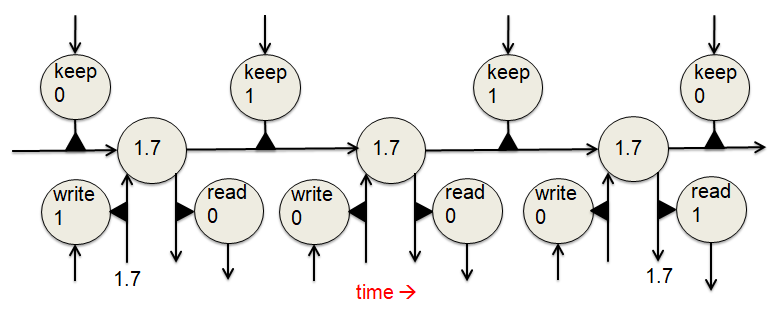
* Again, at the next time step, the keep gate is set to 1, so the information is *stored* for *one more time step*.



* And then, we're going to set the write gate to 0, so *no information is written in*, but we're now going to *retrieve the information* by setting the reed gate to 1.



* The value of 1.7 then *comes out* of the *memory cell* and goes off to *influence the rest of the network*.
* And if we don't need it anymore then the keep gate can be set to 0 and the *information will be removed*.



* Back-propagation: Now, if you look at the **1.7** that *comes out* when we do the *retrieve* and you look at the path *back to the 1.7 that came in*, along that path notice the little ***triangular symbols*** and ***next*** *to each* ***triangular symbol*** *is a* ***1***.
* That means that the ***effective weight*** on that *connection* is a **1**. So as we *go back* along that path, whatever error derivative we have for the **1.7** when it's retrieved, gets ***back-propagated*** *to the 1.7* when it's stored.
* So if you'd rather retrieved a bigger value to make the right things happen now, you can *send the information back* and tell it, it should have stored a ***bigger value***.
* Also notice that as long as the relevant gates have values of **1**, there's *no attenuation* in this *back-propagated signal*. It's got just the properties we want.
* Of course if they're logistic gates there will be some ***slight attenuation***, but it can be very small and so information can travel back through hundreds of time steps.
* **Reading cursive handwriting**

**RNN** with **LSTM** is very good at reading cursive handwriting. It's a very natural task for RNN.

* The input is just a ***sequence o***f the x and y coordinates of the *tip of the pen*, and some information about whether the *pen is on the paper or not*.
* i.e. the input is a sequence of (x, y, p) coordinates of the tip of the pen, where p indicates whether the pen is up or down.
* The output is going to be a sequence of recognized characters.
* Graves & Schmidhuber in 2009, showed that RNNs with LSTM are extremely good at this task.
* They're currently the best systems for reading cursive writing and Canada Post is starting to ***use them*** for ***reading handwriting***.
* Graves & Schmidhuber in 2009, ***didn't*** use *pen coordinates* as input. They used a *sequence of* ***small images***. And that means they can deal with ***optical input*** where the ***timing*** *of the pen is unknown*.
* They can look at images after they've been written and read them.
* **A demonstration of online handwriting recognition by an RNN with Long Short Term Memory (from Alex Graves)**

Following is a demonstration of Alex Graves's system working on pen coordinates. There are *4 streams of information*.

* Row 1: The ***top row*** shows *when the characters are recognized*.
* The system *never revises its output* so difficult decisions are more delayed.
* So if it has to make a difficult decision, it *delays it for a little bit*, so that it can see a *little distance into the future* to help it ***resolve*** ***ambiguities***.
* Row 2: The ***second row*** shows the ***states*** of a ***subset*** *of the memory cells*, and you should notice how they *get* ***reset*** when it *recognizes* a character.
* Row 3: The ***third row*** shows the *actual writing* and all the net sees is the *coordinates of the tip* of the pen. Just two numbers plus some information about whether the *pen* is ***up*** or ***down***.
* Optical input actually works a bit better than pen coordinates.
* Row 4: Finally, the ***fourth row*** shows something much more complicated. It shows the ***gradient backpropagated*** all the way to the locations, i.e. the way to the **x** and **y** inputs from the *currently most active character*. This lets you see which bits of the data are influencing the decision.
* So what you get to see is, for the most active character.
* If you ***backpropagate*** *from that character* and ask what would make that *most active character more active*, you get to see which bits of the input are *affecting the* ***probability*** that it's that character.
* So that let's you see how the decisions, are depending on things that happened in the past.
* Following is the movie: Notice the 4th row, it only shows the ***currently most active characters***.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

