Chapter – 8

**Neural Networks for Machine Learning**

**Geoffrey Hinton**

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

**Hessian free optimization**

Lectures: Geoffrey Hinton

A brief overview of Hessian free optimization

Modeling character strings

Predicting the next character using HF

Echo State Networks

**WARNING: OPTIONAL EXTRA MATERIAL**

The material in this chapter is considerably more difficult than in most of the other chapters. We have included it for those who want to get some idea of how the HF optimizer works.

**8.1 A brief overview of Hessian free optimization**

In this section we'll get a brief overview of the Hessian-Free optimizer that can be used to train RNN very effectively. It's very complicated optimizer and so we're not going to the details about this optimizer.

* We're going to have a general feel for how it works. And then in the *next section*, we'll see its performance on an interesting problem.
* **How much can we reduce the error by moving in a given direction?**

When we're training the weights of a neural network, we are trying to get as far down ***the error surface*** as possible.

* **So one question is:** *If we choose a direction to move in and we keep going in that direction (just the right distance), how much does the error decrease (reduction in the error) before it starts rising again? We assume the curvature is constant (i.e. it’s a quadratic error surface).*
* We can assume that the ***magnitude*** *of the gradien*t ***decreases*** as we move ***down*** the gradient. Assuming that the error surface is concave upward like a bowl (i.e. the error surface is convex upward).
* *The maximum* ***error reduction*** *depends on the* ***ratio*** *of the* ***gradient*** *to the* ***curvature****:* The ***maximum error reduction*** that we can get by going in a *particular direction* depends on the *ratio* of the *gradient to the curvature*.
* So we want to move in directions that have a good ratio. A good direction to move in is one with a high ratio of gradient to curvature.
* Even if the ***gradient*** itself is quite ***small***, we want the ***curvature*** to be even ***smaller***.

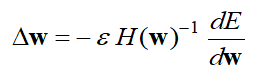
|  |  |
| --- | --- |
| * Consider this example of a *direction* we could *move in* where the ***vertical axis*** corresponds to the ***error***, the ***horizontal axis*** corresponds to the ***weights*** in the direction we're moving in, and the ***blue arrow*** corresponds to the ***reduction*** we get if we start at that ***red point***. |  |

|  |  |
| --- | --- |
| * Here's another example of a surface that has a *gentler gradient* but because it's got a *better ratio* as the *gradient* to the *curvature*, we get a ***bigger reduction*** in the ***error*** by the time we get to the ***minimum***. * The question is, *how can we find directions like this second one?* Directions, in which even though the gradient may be small, the curvature is even smaller. |  |

* **Newton’s method**

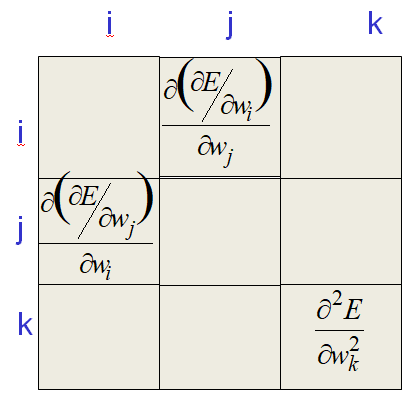
Newton's method addresses the basic problem with steepest descent on a quadratic error surface, which is that the ***gradient******isn't*** *the* ***direction*** that we want to go in.

* If the error surface has *circular cross-sections* and is quadratic, the gradient is a good direction to go in. It will ***point*** straight to the ***minimum***.



* The idea of Newton's method is: to apply a ***linear transformation*** that turns ***ellipses*** into ***circles***. If we apply that *transformation* to the ***gradient vector***, it will be as: *"if we were going downhill in a circular error surface"*.
* To do this, we need to multiply the ***gradient*** by the *inverse* of the *curvature matrix*. So is the ***curvature matrix***, sometimes called the ***Hessian*** (it's the function of the weights, i.e. , and we're taking it's inverse ).
* After taking its ***inverse*** and multiply by the ***gradient vector*** (i.e. ) we need to go *some distance* in that *direction*.
* *On a real quadratic surface it jumps to the minimum in one step:* If it's a truly quadratic surface and we choose epsilon correctly (which is quite easy to do), we'll *arrive* at the *minimum of the surface* in a ***single step***.
* Of course, that single step involves something complicated, which was inverting that Hessian matrix.
* Unfortunately, with only a million weights in our neural network, the curvature matrix, the Hessian, will have a trillion terms, and it is totally ***infeasible*** to ***invert*** it.
* **Curvature Matrices**

The curvature matrices look like below.



* Each element in the curvature matrix specifies how the *gradient* in *one direction* *changes* as we move in some *other direction*. i.e. each weight, or tells you how the *gradient in one direction changes* as you change in *another direction*.
* In other words, as we change , how does the *gradient of the error* w.r. to change? That's what a typical *off diagonal term* tells you.
* The *terms* ***on*** *the* ***diagonal*** tell you how the *gradient of the error* changes *in the direction of a weight* as you change that weight.
* The *off-diagonal terms* in a curvature matrix correspond to twists in the *error surface*.
* A twist means, when you *travel in one direction*, the *gradient in another direction changes*. If we have a nice *circular bowl*, all those *off-diagonal terms* are *zero*. As we travel in one direction, the gradient in other directions *doesn't change*.
* The reason ***steepest descent goes wrong*** (when you have an elliptical error surface) is that the *gradient for one weight* gets ***messed up*** by the *simultaneous changes* to all the *other weights*.
* i.e. as we travel in one direction, the gradient in another direction changes.
* so if we *update one of the weights*, at the same time as we're *updating all the other weights*, *all those other updates* will cause a ***change*** in the ***gradient*** for the *first weight*.
* That means, when we update it, *we may actually make things* ***worse***. The *gradient* may have actually *reversed sign* due to *all the changes* in the *other weights*.
* So as we get more and *more weights*, we need to be *more and more cautious* about *changing each one of them*, because the ***simultaneous*** ***changes*** in all the *other weights* can change the gradient of a *weight*.
* The curvature matrix determines the size of those interactions.
* **How to avoid inverting a huge matrix**

We have to deal with the curvature without actually inverting the matrix, because the *curvature matrix* has *too many terms* in a big neural net.

* One thing we can do is to just look at the ***leading diagonal*** *of the curvature matrix* and make our *step size* ***depend*** on that *leading* *diagonal*.
* Maybe we can get some benefit from just *using* the *terms* along the *leading diagonal* (Le Cun); it will get us to make *different step sizes* for *different weights*,
* But the *diagonal terms* are only a *tiny fraction of the interactions* (they are the self-interactions), so we're ***ignoring*** most of the ***terms*** in the *curvature matrix* when we just look at the ***leading diagonal***. In fact, we're ignoring nearly all of them.
* Another thing we could do is, try an ***approximate*** of the ***curvature matrix***, using a *matrix of much lower rank* that captures the *main aspects* of the ***curvature matrix***.
* The curvature matrix can be approximated in many different ways: We can use different methods like: *Hessian-Free* methods and *LBFGS*, and many other methods that try and do an *approximate second order method for minimizing the error*.
* In the Hessian-Free Method, we make an *approximation* to the *curvature matrix* and then we *assume* that the *approximation is correct*.
* So we assume: we know "*what the curvature is*" and that the "*error surface really is quadratic*". And then, starting from wherever we are now (on the surface), we *minimize the error* using an efficient technique called conjugate gradient.
* Once we got close to a *minimum* on *this approximation to the curvature*, we then make *another approximation* to the *curvature matrix* and we use conjugate gradient to *minimize again*.
* For RNNs it's *important* to add a penalty for *changing* any of the ***hidden activities*** *too much*.
* That penalty will prevent us for example, from *changing a weight early* on that causes *huge effects later on* in the sequence.
* We don't want to get *effects* that are *too big*, and if we look at the *changes* in the *hidden activities* we can prevent that by *penalizing* those changes.
* If we put a ***quadratic penalty*** on those changes, we can combine that with the rest of the *Hessian-Free method*.
* **Conjugate gradient**

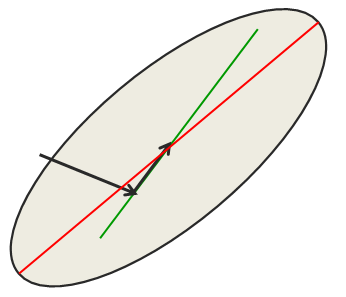
There is an alternative to going to the *minimum in one step* by multiplying by the *inverse of the curvature matrix*. Conjugate gradient is a very clever method that, *instead of trying to go* ***straight*** to the *minimum* like in Newton's method, tries to *minimize* in *one direction* at a time.

* So it starts-off by taking the direction of: *steepest descend* and goes to the *minimum* in that direction.
* That might involve *re-evaluating the* ***gradient*** or *re-evaluating the* ***error*** a few times to find the minimum in that direction.
* Once its done that, it now finds another direction and goes to the minimum in that second direction. I.e. it uses a *sequence of steps* each of which finds the *minimum* along *one direction*.
* ***Conjugate Direction:*** The clever thing about the technique is, it chooses its *second direction* in such a way that it *doesn't mess up the* *minimization* it already did in the *first direction*. That's called a conjugate direction.
* Conjugate means that as you go in the *new direction*, you *don't change the* ***gradients*** in the *previous directions*. It's a funny idea. It's like the idea of a twist in an *error surface*.
* So it makes sure that each *new direction* is "conjugate" to the *previous directions* so you do not mess up the minimization you already did.
* A twist means when you go in *one direction*, you change the gradient in *another direction*.
* And a conjugate direction is one you can go in, that in a sense, it *doesn't have a twist*. You go in that direction and the gradient in the first direction doesn't change.

|  |  |
| --- | --- |
| * **A picture of conjugate gradient**   Following is a picture of an ellipse and the red line is the major axis of the ellipse. We start off by doing *one step* of *steepest descent* all the way to the ***minimum*** in that ***direction***.   * Note that the ***minimum*** *won't actually lie* on the ***red line***. |  |

|  |  |
| --- | --- |
| * ***On*** the ***red line***, the *gradient will be zero*, at *right angles* for that *red line*, because it's the *bottom* of the *ravine*. * But the direction we're going in, *isn't actually at* ***right angles*** *to that point*. We can make a little bit more progress by making a ***small step*** *at* ***right angles*** *to the red line* and then a *small step* ***along*** *the red line*. * Since the ***red line*** *slopes down* towards the *middle of the ellipse*, that's going to make some progress for us. * So when we ***minimize*** in the ***first direction***, we'll go *slightly across the bottom of the ellipse*. |  |
| * And when we reach that point that's a minimum, there's an interesting property of all the points that lie on the green line: * On that green line, the *gradient* in the *direction* of that *black arrow* is zero. * So we can go anywhere *along that green line* and we won't *destroy* the fact that we are at a *minimum* in the *direction of the black arrow*. * i.e. the *gradient* in the *direction of the first step* is zero at ***all points*** on the ***green line***. So if we *move along the green line* we don’t mess up the *minimization* we already did in the *first direction*. |  |

* If we can keep doing that from many directions in a *high dimensional* ***error surface***, we'll eventually be at a ***minimum*** in ***many different*** ***directions***. And if we are at a *minimum in as many different directions as there are dimensions in the space*, we'll be at the global minimum.
* So, we take the first step of steepest descent, we then figure out the *direction of that green line*, and then, we do a *search along the green line* to find how far we should go in order to *minimize* *the* ***error*** *along the* ***green line***.
* And we take our second step, like this:



* And now, in this *2-dimensional space (in this example)*, we'll be at the ***minimum***. Because, we're at the *minimum* in the *direction of the first step* and we're now at a *minimum* in the *direction of the second step*, While still being at a minimum in the direction of the first step and so that must be the global minimum.
* **What does conjugate gradient achieve?**

*What conjugate gradient achieves is that:* it gets to the global minimum of an N-dimensional quadratic surface in only N steps. It's very efficient. It does that because it manages to get the *gradient to be zero* in *N different directions*.

* After N steps, *conjugate gradient* is *guaranteed* to find the *minimum* of an ***N-dimensional quadratic surface***. Why?
* They're not orthogonal directions, but they are *independent of one another* and so that's efficient to be at the global minimum.
* More importantly, in ***many less than N steps*** on a *typical quadratic surface*, it will have reduced the *error* very *close to* the *minimum* *value*, and that's why we use it.
* So, we're *not going to do the* ***full N-steps***, that would be as expensive as inverting the whole matrix. We're going to do ***many less than N steps***, and we're going to get quite close to the minimum.
* You can apply *conjugate gradient* *directly* to a *non-quadratic error surface*, like the *error surface* for a ***multilayer non-linear neural net*** and it usually works quite well.
* It's essentially a batch method, but you can apply it to large mini batches. And when you do that, you do ***many steps of conjugate gradient*** on the *same large mini batch* and then you move on to the *next large mini batch*. That's called non-linear conjugate gradient.
* The Hessian-Free optimizer uses *conjugate gradient* for minimization on a *genuinely quadratic surface* and that's what conjugate gradient is best at. It works much better for that than for a ***non-linear surface***.
* *The* ***genuinely quadratic surface*** *is the* ***quadratic approximation to the true surface****:* This genuinely quadratic surface that HF is using it for is: the *quadratic approximation to the true surface* that was made by the Hessian-Free method.
* So it makes that *approximation*, it uses *conjugant gradient* to get close to a *minimum*, for the *first approximation*. And then it makes a new approximation to the curvature, and does it again.

**8.2 Modeling *Character Strings* with Multiplicative Connections**

We're now going to apply Hessian-free optimization, to the task of *Modeling character strings from Wikipedia*.

* ***The idea is:*** *read a lot of Wikipedia* and then try to *predict* the *next character*.
* Before we get to see what the model learns, we'll learn about:
* *why we need multiplicative connections* and
* *how we can implement those* *multiplicative connections* efficiently in a recurrent neural network (RNN).
* **Modeling text: Advantages of working with characters**

***One question is:*** Why we chose to model character strings rather than strings of words (which is what you normally do when you're trying to model language)?

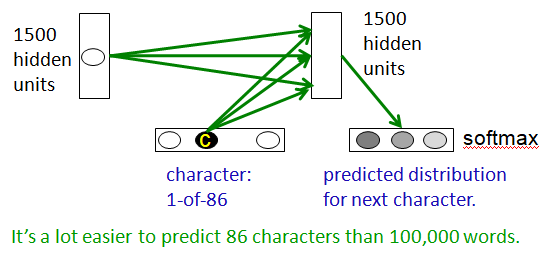
* The web is composed of character strings. Any learning method that's powerful enough to *understand* what's *going on in the world* by reading the web, ought to find it trivial to *learn which strings make words*. (As we'll see, this turns out to be true.)
* We want something that will ***read Wikipedia*** *and understand the world*.
* ***Pre-processing text*** *to get words is a* ***big hassle****:* If we have to pre-process the text in Wikipedia into words, it's going to be a big hassle. There's all sorts of problems:
* Morphemes: The first problem is morphemes. The *smallest units of meaning*, according to linguists, are morphemes. So we're going to have to *break up a word* into these *morphemes* (prefixes, suffixes etc) if we want to deal with it sensibly.
* The problem is, it's not quite clear what morphemes are. There's things that are a bit like morphemes, but that a linguist wouldn't call a morpheme.
* Subtle effects: In English, if you take any word that starts with the letters sn, it has a very *high chance of meaning* something to do with the lips or nose, particularly the *upper lip* or *nose*. So the words like snarl, and sneeze, and snot, and snob, and snort. There's too many of these words for it just to be coincidence.
* Many people say: *Yes, but what about snow? That's got nothing to do with the upper lips or nose.* But, ask yourself something, why is *snow* such a good word for *cocaine* (now its related to nose)?
* ***Words*** *that come in* ***several pieces:*** There's words that come in several pieces. So normally, we'd want to treat New York as *one lexical item*. But if we're talking about the ***New York minster roof***, then we might want to treat New and York as *two separate lexical items*.
* *Languages like* ***Finnish*** *and* ***German****:* There's languages like Finnish. Finnish is an agglutinative language, so it puts together *lots of* *morphemes* to make *great big words*. So here's an example of a word in Finnish that *takes about five words in English* to say the same thing: ymmartamattomyydellansakaan



* **An obvious recurrent neural net**

Following is an obvious kind of RNN we might use to *try* and *model character strings*. It has a hidden state and in this case, we're going to use 1500 hidden units.

* The hidden state dynamics is:
* The hidden state at time T provides inputs to *determine* the hidden state at time T+1.
* The character also provides some inputs.
* So we add together the *effect of the current character* with the *previous hidden state* to get the *new hidden state*.



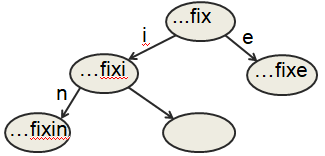
* When we arrive at a *new hidden state*, we *try and predict* the next character.
* We have a single Softmax over the 86 characters, and we get the hidden state to *try and assign* ***high probability to the correct next*** ***character***, and ***low probability*** *to the others*.
* We ***train*** the whole system by backpropagating from that Softmax (the lower probability of getting the correct character).
* We backpropagate that through the ***hidden to output*** connections back through the ***hidden to character*** connections, and then back through the ***hidden to hidden*** connections, and so on and all the way back till the beginning of the string.

It's a lot easier to predict 86 characters than 100,000 words. So it's easier to use a ***Softmax*** *at the output*, we don't have the *problem of having a great, big Softmax*.

* **A sub-tree in the tree of all character strings**

Now let's explain why we *didn't use* that kind of recurrent net, but instead used a *different kind of net* that worked quite a lot better.

* You could *arrange all possible character strings* into a tree with a branching ratio of 86, in our case.



* It's a tiny little *subtree*, of that great *big tree*. In fact, this little *subtree* will *occur many times*, but with different words, those other characters/words are represented by that "**…**" before the **fix**.
* So this represents that we had a *whole bunch of characters*, then we had **f**, **i** and then **x**.
* And each time we get a new *character*, we move *one step down* in this tree to a *new node*.
* For example, if we get an i, we're going to the left.
* If we get an e, we're going to the right, and so on.
* ***There's exponentially many nodes:*** There's exponentially many nodes in the tree of all character strings of length N. So this is going to be a very big tree. We couldn't possibly store it all.
* If we could store it all, what we'd like to do is: put a probability on each of those *arrows*. It's the ***probability*** *of producing that* ***letter****, given the* ***context*** *of the node*.
* ***How it's done with RNN:*** In an RNN, *each node* is a *hidden state vector*. The *next character* must transform this to a *new node*.
* We try and deal with the fact that the *full tree is enormous* by using a hidden state vector to represent each of these nodes.
* So now, what the next character has to do is: take the ***hidden state vector*** that's *representing the whole string of characters followed by* **fix** and operate on the *hidden state vector* to ***produce*** the appropriate *new hidden state vector* if the next character was an **i**.
* So when we see an **i**, we want to turn the *hidden state vector* into a *new hidden state vector*.
* If the nodes are implemented as *hidden states* in an RNN, different nodes can *share structure* because they use distributed representations.
* It's a nice thing about implementing these nodes in this *character tree* by using the hidden state of RNN, is that we can share a *lot of structure*.
* For example, by the time we arrive at that node, that says **fix**, we may have decided that it's probably a verb.
* And if it's a verb, then **i** is quite likely because of the ending **ing**. And that knowledge that **i** is quite likely with a verb, can be shared with *lots of other nodes* that don't have **fix** in.
* So we can get **i** to operate on the part of the state that represents that it's a ***verb***, and that can be *shared between all the verbs*.
* The *next hidden representation* needs to depend on the *conjunction of the current character* and the *current hidden representation*.
* Notice that, it's really the ***conjunction of the current state*** we're at and the ***character*** that determines where we want to go. We ***don't want*** **i**, to give us a state that's expecting to get an **n** next if it ***wasn't a verb***.
* So, we don't want to say that **i** tends to make you expect an **n** next. We really want to say, if you ***already*** think it's a ***verb***, then when you see an, **i**, you should expect an **n** next.
* It's the ***conjunction*** of the ***fact*** that we think ***it's a verb***, and that ***we saw an*** **i**, that gets us into this state labeled **fixi**, that's ***expecting to see*** an **n**.
* **Multiplicative connections**

We're going to try and capture that fact by using *multiplicative connections*.

* Instead of using the *character inputs to the recurrent net* to provide *extra additive input* to the *hidden units*, we're going to use those characters to swap-in a whole *hidden-to-hidden weight matrix*.
* We could use the *current input character* to choose the whole hidden-to-hidden weight matrix. The character is going to determine the *transition matrix*.
* ***Requires 86x1500x1500 parameters:*** Now, if we did that in the naive way, we'd have each of the 86 characters to find a 1500x1500 matrix and that would be a lot of parameters.
* ***This could make the net overfit:*** If we have that many parameters, the net likely to overfit, unless we run it on *huge amount of text*, for which we might not have time.
* ***So the question is***, *can we achieve the same kind of multiplicative interaction*? Where the *character* determines the *hidden-to-hidden weight* *matrix* using fewer parameters, by making use of the fact that *characters have things in common*?
* For example, all of the digits are all quite similar to each other in the way in which *they make the hidden state evolve*.
* So, we want to have a *different transition matrix* for each of those *86 characters*, but we want those *86 character-specific weight matrices* to *share* ***parameters*** (that's a reasonable thing to do because we know that characters 8 & 9 should have *very similar transition matrices*).
* **Using factors to implement multiplicative interactions**

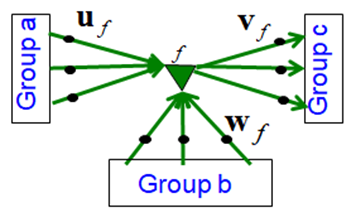
So here's how we're going to do it. We're going to have things called factors, and they're going to be denoted by this little triangle with an f above it. That factor means: Group a and Group b interact *multiplicatively* to provide input to Group c.

* We can get groups a and b to interact multiplicatively by using “factors”.

|  |  |
| --- | --- |
| * What each factor does is: It first computes a weighted sum for each of its input groups (in our case two input groups). * We take the *vector state* of Group a, (we just call it a), and we multiply that by the *weights on the connections* coming into the *factor*. * In other words, we take the scalar product of the vector a and the weight vector u, and that gives us a number at the *left hand vertex* of that triangle. |  |

* Similarly, we take the *vector states* of Group b and we multiply it by the weight vector w, and we get another number of the *bottom* *vertex* of the triangle.

|  |  |
| --- | --- |
| * Then it sends the product of the weighted sums to its output group. * That is, we multiply those two numbers together and that gives us a number (or scalar). And we use that *scalar* to *scale* the outgoing weights v in order to provide input for Group c. * So the input to Group c is just the *product of the two numbers* that come into the *two vertices of the triangle*, times the outgoing weight factor v. |  |



* We can write that as an equation. The input that factor f provides to Group c, i.e. its ***vector of input*** to Group c, , is a scalar input to f from Group b (that's got by *multiplying the state* of Group b by the weights w, i.e.) times a scalar input to f from Group a (that's got by *multiplying that state* of Group a by the weights u, i.e. )

|  |  |
| --- | --- |
| scalar input to f from Group b =  scalar input to f from Group a = |  |

* We then take the product of those two scalars and and multiply the weight vector by that, and that's the input that the factor f gives to Group c.
* Then, of course, we're going to have a whole *bunch of those factors*.
* **Using factors to implement a set of basis matrices**

There's another way we can think about those factors that gives more insight into what's going on. *Each* of the *factors* actually defines a very simple kind of *transition matrix*. It's a transition matrix that has rank 1.

* i.e. Each factor defines a rank 1 transition matrix from a to c.

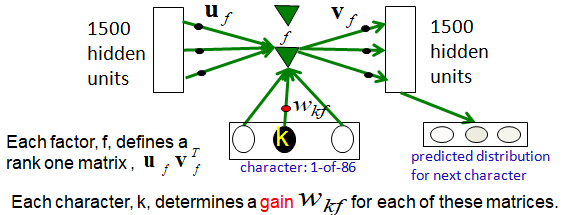
|  |  |
| --- | --- |
| * The equation we had previously *treats a* ***factor*** *as* computing two scalar products, multiplying them together, and then using that as a weight on the outgoing vector v. * We can rearrange that equation, so that we get one scalar product, and then we rearrange the last bit so that now, we take the outer product of the weight vector u and the weight vector v, and that gives us a matrix. * And the *scalar product* that would be computed by *multiplying* b by w is just a *coefficient* on that matrix (scalar coefficient). So we get a scalar coefficient . |  |

* We multiply a rank 1 matrix by that scalar coefficient to give us a *scaled matrix*. And then, we multiply the *current hidden* *state* by this *scaled matrix* to determine the input the factor f gives to the *next hidden state*.

|  |  |
| --- | --- |
| * If we sum that up over all the factors, the total input to Group c is just a sum over all *factors of a scalar* times a *rank 1 matrix*, and that *sum* is a great *big matrix*, that's the transition matrix, and it gets multiplied by the current hidden state to produce the new hidden state. |  |

* So, we can see that we *synthesized* the *transition matrix*, actually, these *rank 1 matrices* provided by *each factor*.
* And what the *current character* in *Group b* has done is, is it's ***determine*** the ***weight*** on each of these *rank 1 matrices*.
* determines the *scalar weight*, a *scalar coefficient* to put on each of the matrices, which going to compose this great big character specific weight matrix.
* **Using 3-way factors to allow a Character to create a whole Transition Matrix**

Following is the picture of the whole system, we have about 1500 factors. And the *character input* is different, only ***one*** of those is ***active***. So there will only be ***one*** relevant ***weight*** at a time.



* And that *weight* from the *current character* k, we call it .
* Each factor defines a *rank 1 matrix*, the *rank 1 matrix* is got by taking the ***outer product*** of and i.e. .
* The weight is the gain that used on that *rank 1 matrix*. That is, each character, k, determines a gain for each of those rank 1 matrices.

So the character determines the *gain* , you multiply the *rank one matrix* by that *gain*, you add together the scaled matrices for all the different factors and that's your transition matrix.

**8.3 Learning to predict the next character using HF**

In this section, we're going to see what happens when the Hessian-Free optimizer is used to optimize the *RNN* containing *multiplicative* *connections* and the network is *trained* to predict the *next character in Wikipedia*.

* The network is trained on millions of characters and it works remarkably well. It learns a lot about *English* and it becomes very good at *completing sentences* in interesting ways.
* **Training the character model**

**Ilya Sutskever** used 5 million strings of 100 characters each taken from English Wikipedia. For each string, he starts predicting after *eleventh character*.

* So the ***recurrent network*** starts off in a ***default state***.
* It reads eleven characters, changing its hidden state each time.
* And then its ready to start predicting.
* It gets *trained* by *back propagating* the *errors* it makes in *prediction*.
* He used the HF (Hessian-free) optimizer, and it took about a month on a very fast GPU board to get a really good model.
* His model was the best RNN for character prediction and is probably the *best single model* there is for predicting characters.
* We can do better than this model by combining many different models and using a neural network to decide which one to use, but for a single model, Ilya's was the best.
* It works in a very different way, from the best other models.
* Ilya's model can balance *quotes* and *brackets* over *long distances*. Any model that relies on *matching* a *specific previous context* can't do that.
* *Models that rely on matching previous contexts cannot do this:* For example, say it has a bracket, and it wants to close it after 35 characters. In order to do that properly, a *model* that relies on *matching* *previous contexts*, would have to *match all 35 intervening characters*. And it's very unlikely that it has that whole string stored.
* **How to generate character strings from the model**

Once the model is learned, you can see what it knows by ***generating strings*** *from the model*. Of course you have to be very careful not to over-interpret what it says. The way we generate strings is:

* We ***start*** the model with its *default hidden state*.
* Then we give it a *"****burn-in****" sequence* of characters. So we feed it *characters* and let it *update its hidden state* after *each character*. And then we let it start predicting.
* Then we look at the *probability distribution* it *predicts* for the *next character*.
* We pick a character *randomly* from that *distribution*. So if it predicts, the *probability* of Q is 1 in a 1000, we pick Q one times in a thousand.
* We then tell the net that whatever character we picked was the character that actually occurred. And then we ask it to predict the next character.
* In other words, we're telling it whatever it guesses is correct. We let it continue to make characters until we've got as many as we want, and then we *look* at the *strings it produces* to see what it knows.
* So here's an example of a string produced by **Ilya's** network after some burn in. This was selected from a much longer passage of text. But it's a continuous passage, which shows you that it works pretty well.

"

He was elected President during the Revolutionary War and forgave Opus Paul at Rome. The regime of his crew of England, is now Arab women's icons in and the demons that use something between the characters‘ sisters in lower coil trains were always operated on the line of the **ephemerable** street, respectively, the graphic or other facility for deformation of a given proportion of large segments at RTUS**)**. The B every chord was a "strongly cold internal palette pour even the white blade.”

"

* Notice that it has weird semantic associations.

For example: "Opus Paul at rome". No person would ever say that. But we understand that Opus and Paul and Rome are all *highly* *interconnected*. You'll notice it *doesn't* really *have* any long range thematic structure. It pretty much changes topic after each full stop ".".

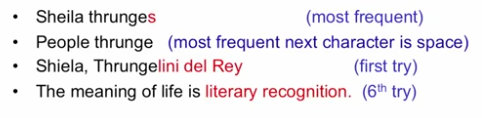
* One amazing thing is that is produces very few ***non-words***. What that means is that, even though it's predicting *probabilities of characters*, as soon as you've got *enough characters*. So there's only one way to complete it as an English word, it will predict the *next* *character* almost *perfectly*.
* If that wasn't the case, it would produce *non-words*.
* Even when it does produce a *non-word*, like the *word in red*, it's a very good non-word. For example ephemerable is much more like an English word.
* You'll also notice it's produced a ***closing bracket*** *without an opening one*. So it doesn't always balance brackets. It just does it quite frequently.
* You'll also notice at the end, that its produced an *opening quote* and then a *closing quote* much later.
* That's consistent behavior on its point. It really did produce that closing quote, because it had an open quote earlier on.
* If you look at this text you can see there's a lot of ***good local syntax***. So, little strings of *three or four words* look perfectly *reasonable*.
* There's also lots of *semantic knowledge*.
* **Some completions produced by the model**

One thing we can do is: we can ***test*** the model by giving it *carefully designed strings* to see what it knows.

* So let's try it by giving a non-word. The word ***thrunge*** "T H R U N G E" is not an English word.
* But most *English speakers*, when they see that word, would expect it to be a *verb* because of it's form.
* So we gave it opening contexts in which that might be a verb, to see what *character* was most likely to *come next*.
* So, if you give it "Sheila Thrunge" and ask for the next character, the most frequent one is an "s", which suggests that the model knows that "Sheila" is *singular*, just from reading Wikipedia.
* If you give it "people thrunge" the most frequent next character is a "space", not an "s", which suggests that the model knows that "people" is *plural*.
* If give it a list of names. We used capitals for the names with a comma in between. And we put a capital T for Thrunge so it looked like a name, to see what it would do with that:

Shiela, Thrungelini del Rey

* So it actually completed it *as a name*. And if you look at the name it made, it's not a bad name. It indicates that it knows about names in many languages.
* You can also give it "The meaning of life is", and the see what comes next.
* If it produced 42, that wouldn't be very interesting because that could be somewhere in Wikipedia.
* It produces some random things. But in it's first *10 tries*, it produced "The meaning of life is literally recognition". Which is syntactically and semantically sensible.



* We then trained the model some more and presented it with "The meaning of life is" again. We took the *first 10 things* it produced and we're going to show you the most interesting one.
* The completion it produced for "The meaning of life is", suggests that by *reading Wikipedia* it really is *beginning to understand* something about the *meaning of life*. That's probably just wild over-interpretation, though. So here's its completion:

The meaning of life is the tradition of the ancient human reproduction: it is less favorable to the good boy for when to remove her bigger.

(one of the first 10 tries for a model trained for longer).

* **What does it know?**

So, what does the model know after it's read all these characters in Wikipedia?

* It certainly knows about *words*. It produces almost always, *English words*.
* It will produce *strings of initials*, typically in *capitals*. It can produce numbers and dates and things like that.
* But it doesn't produce *non-words* very often. It produces *non-words* extremely rarely. And when it does produce them, they're typically very *plausible non-words*.
* It also knows a lot about proper *names*, like Frangelini Del Rey.
* It knows about *dates* and *numbers*, and the context in which they occur.
* It's good at *balancing quotes* and *brackets*, and in fact it can actually *count brackets*.
* If you give it *no opening brackets*, it's very unlikely to produce a closing bracket.
* If you give it *one opening bracket*, it's quite likely to produce a closing bracket in the *next twenty characters* or so.
* If you give it *two opening brackets*, it'll produce a closing bracket very quickly. Giving it three doesn't seem to make it any faster.
* It clearly knows a lot about syntax because it's able to produce *sensible little strings of English words*. But its very *hard* to *pin-down* exactly what form this knowledge has.
* It's not like trigram models which have just learned *little sequences of words*.
* Or rather they have a table that contains *little sequences of words*.
* *Its syntactic knowledge is not modular:* It's actually ***synthesizing*** *strings of words*. And it's synthesizing them with *sensible syntax*. It's very hard to say though what ***form*** that *syntactic knowledge* had.
* It's not a bunch of rules like a linguist has. It's much more like what's in the linguists head when he speaks a language.
* It knows a lot of *weak semantic associations*. So, for example, it only ever produced the word "Wittgenstein" once. And it produced that soon after producing the word "Plato". So it knows that Plato and Wittgenstein are associated.
* It clearly knows that *cabbage* is associated with *vegetable*. It doesn't know much about the *precise ways* in which these things are associated. People are like that too if you get them to respond very fast.
* For example: If we asked a question, and we shout out the answer, and we have to shout it out really fast. We get rewarded for responding very quickly. It doesn't matter what we say.
* So the question is, "what do cows drink?".
* Most people, when given that question, shout out milk. Now, most cows don't drink milk most of the time.
* We say milk because it's associated with both drink and with cow. But it's not logical to say milk.
* **RNNs for predicting the next word**

Recently, *Thomas Mikolov* and his collaborators have been training *large RNNs* to predict the *next word* in *large datasets*. They use the same technique as the feed-forward neural nets.

* It first converts a word to a real valued feature vector.
* Then use those feature vectors as input to the rest of the network.

And they do better than the feed forward neural nets.

* They trained quite large RNNs on quite large training sets using BPTT, i.e. Recurrent neural networks leverage backpropagation through time (BPTT) algorithm.
* They do better than feed-forward neural nets.
* They do better than the best other models.
* They do even better when averaged with other models.
* One interesting property of the RNNs is that they require *less training data* than the *other methods* to reach a given level of performance.
* More importantly, as the *data sets get bigger*, the RNNs *improve faster* than the other methods.
* For example: methods like *trigrams*, do get *better with bigger data sets* but it's a very *slow process*. You need to double the size of the data set to get a small improvement.
* With *RNNs*, they can make much *better* *use of the data*. This means it's going to be very hard to beat them as data sets get bigger. I think it may be the same story as for *object recognition* with *large, deep neural nets*.

Once the neural nets get ahead, they can make better use of faster computers and bigger data sets. And so, it's going to be very hard for other methods to catch up.

**8.4 Echo State Networks**

In this section we're going to discuss the eco-state networks. These *eco-state networks* use a clever trick to make it much easier to learn a RNN. They *initialize* the *connections* in the RNN in such a way that it has a *big reservoir* of ***coupled oscillators***.

* So if you provide input to it, it *converts* that *input* into the states of these oscillators, and then you can *predict* the *output* from the *states of these oscillators*.
* The only thing you have to learn is how to *couple the output* to the *oscillators*.
* It resolves the problem of *learning* ***hidden to hidden*** connections or even ***input to hidden*** connections.
* However, to get these networks to be good at complicated tasks, you need a *very* ***big hidden*** *state*.
* At the end of the section we'll see,
* How to use the *initialization* that was carefully designed for ***echo state networks***,
* Use *Back-Propagation Through Time with Momentum* (BPTT with momentum) to train the networks to be even better.
* **The key idea of echo state networks (PERCEPTRONS again?)**

One interesting and quite recent idea about training RNNs, is to *not train the hidden to hidden connections at all*, but to just *fix them randomly*, and hope that you can ***learn sequences*** by just *training* the way they *affect the outputs*.

* This has strong similarities with *old ideas* about perceptions.
* ***Make the early layers random and fixed:*** So a very simple way to train (learn) a *feed-forward neural network*, is to make the *early layers of feature detectors* just be random and fixed.
* You put in sensible sized random weights.
* Then all you *learn* is the *last layer* so that you're learning a ***linear model*** from the *activities of the hidden units* in the *last layer to the outputs*. Since it's much faster to learn a *linear model*.
* i.e. we just learn the *last layer* which is a *linear model* that uses the *transformed inputs* to predict the *target outputs*.

|  |  |
| --- | --- |
| * *A* ***big random expansion*** *of the input vector can help:* A *big*, *random expansion* of the *input vector*, can often make it easy for a *linear* *model* to fit the data (when it couldn't fit the data well, just looking at the raw inputs). * Through the little neural network here, those red weights will be fixed at random. They would *expand* the *input vector* and then using that ***expanded representation***, we try and fit a linear model. * This actually has some quite strong similarities with ***support vector machines (SVM)***. Which are really efficient way of doing this kind of thing. So those same ideas, many years later, were recycled for ***RNNs***. |  |

* The equivalent idea for RNNs is to make the *input to hidden connections* and the *hidden to hidden connections* have ***random values*** that are carefully chosen. And only ***learn*** the *final layer of hidden to output* connections.
* The learning is then very simple if you use ***linear output units***. And it can be done extremely fast.
* This approach is only ever going to work if you *set* the *random connections* very *carefully*, so that the RNN doesn't ***die-out*** with no activity and doesn't ***explode***.
* **Setting the random connections in an Echo State Network**

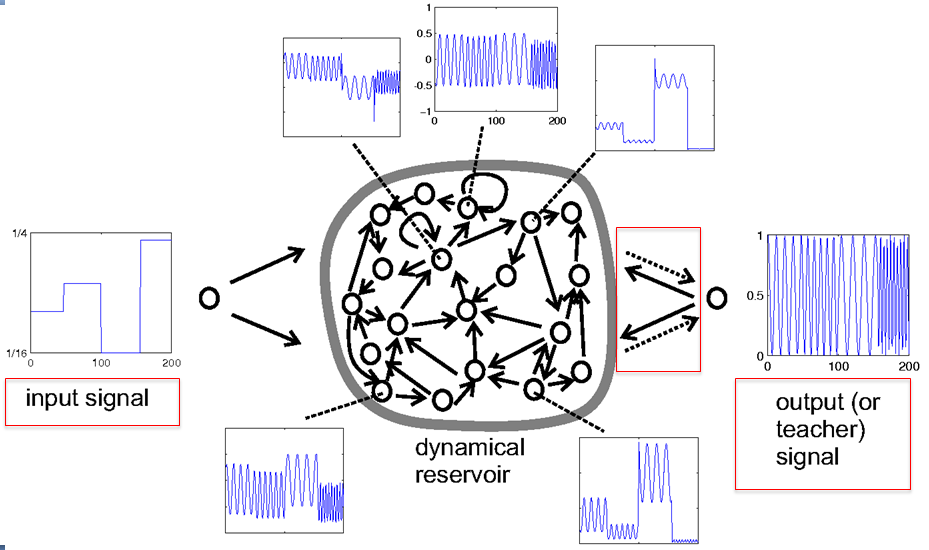
The way to set the random connections in a echo state network is:

* Set the *hidden to hidden weights* so that the *length* of the *activity vector* stays about the *same* after each iteration. For those of you used to *linear systems* and *matrices*, you're setting it so the *spectral radius* is 1.
* i.e. The biggest *eigenvalue* of the matrix of *hidden to hidden weights* is 1. Or it would be 1 if it was a *linear system*.
* And we want to achieve the same property in a *non-linear system*.
* *This allows the input to echo around the network for a long time:* If you set those weights to be about the *right magnitude*, then an *input* can *echo around* in the *recurrent state* for a long time.
* *Use sparse connectivity (i.e. set most of the weights to zero):* It's also important to use *sparse connectivity*. So instead of having lots of *medium size weights*, we have a *few* quite *large weights*. And *nearly all* the *weights* are *zero* in the ***hidden to hidden*** connections.
* What this does is: it makes a lot of *loosely coupled oscillators*. So information can hang around in one part of the net *without* being *propagated* to other parts of the net too quickly.
* *Choose the scale:* It's also important to choose the *scale* of the *input to hidden connections* very carefully.
* Those ***connections*** need to ***drive*** the ***states*** of the *loosely coupled oscillators* but, they ***mustn't*** *wipe* out *information* that those oscillators contain about the *recent history* (i.e. without wiping out the information from the past that they already contain).
* *Experiment frequently:* Fortunately the learning is very fast in echo state networks so we can afford to experiment with the scales of the important connections. We can try many different scales for the *weights* and *sparsenesses*.
* You could think of it as a ***little learning loop*** that's just *learning* the *scales* of those *connections* and it's doing it by sort of feedback that involves the ***experiment***.
* It also helps to *learn* the *level of sparseness* that's needed in the *hidden to hidden* connections, and again because the learning is so fast, you can afford to experiment with that.
* It's often necessary to do those experiments to get the system to work well.
* **A simple example of an echo state network**

Let's discuss a simple example taken from the web of an eco-state network.

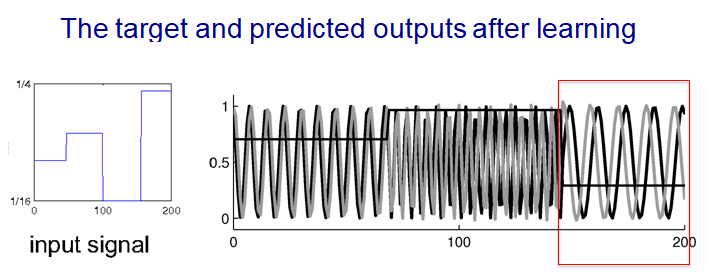
* ***Input Sequence:*** It has an *input sequence* which is a *real value* that ***varies*** with *time*, and specifies the *frequency* of a sine wave for the output of the eco-state network.
* We'd like to *generate sine waves*, and the *input* is gonna *specify the frequency*.
* ***Target output Sequence:*** The *target output sequence* is going to be the *sine wave* with the frequency specified by the input.
* ***Learning Method:*** And it's going to be *learned* simply by *fitting a linear model* that takes the *states of the hidden units* and from those tries to *predict* the correct single *scalar output* value.
* **Example from Scholarpedia**

Following is a picture taken from Scholarpedia of an *echo state network* doing this problem,



|  |  |
| --- | --- |
| * ***Input:*** The input signal is the *desired frequency of the* ***sine wave***. | * And the stuff in the middle is a *big dynamical reservoir*, so that the *inputs* coming from the input signal *drive* those loosely coupled *oscillators*, and cause *complicated dynamics* that *goes on* for a long time.      * And those output weights are *learning to map* that *complicated dynamics to the particular dynamics* you want for the output. |
| * ***Output:*** The output signal after it's learned, or to teach a signal, while it's learning, is a *sine wave* with the frequency *specified by* the *input*. |

|  |  |
| --- | --- |
| * All the other pictures (of the reservoir) are showing you the actual dynamics of *individual units* inside the dynamical reservoir. | * One thing to notice is that there are also *connections from the* ***output*** *back to the* ***reservoir***. Those aren't always needed, but they help to *tell the reservoir* what outputs has been *produced* so far.      * ***The Target and Predicted Outputs after learning:*** Here's the output (what they system actually produces after it's learned), and we can see that at the beginning it's producing a *sign wave*, in *phase*. But at the end, it's producing a *sign wave* of the *right frequency*, but the *phase is wrong*. * That's because we weren't telling what *phase* the *sign wave* should be in. So it's satisfying the requirements of producing an *appropriate* *frequency*. |



* **Beyond echo state networks**

**Good aspects of ESNs**

* Echo state networks can be *trained* very *fast* because they just fit a *linear model*.
* They also demonstrate how important it is to *initialize* the *hidden-to-hidden weight* sensibly.
* And they can do quite impressive modeling of *one dimensional time-series*. That's where they excel. They can look at a *time series* for awhile, and then *predict* it very well a *long time* into the *future*.

**Bad aspects of ESNs**

What they're not so good at is *modeling high dimensional data*, like *frames of acoustic coefficients*, *pre-processed speech*, or *frames of* *video*.

* In order to model data like that, they need *many more hidden units* for a given task, than an RNN that learns/train the *hidden to hidden* connections.
* Recently, **Ilya Sutskever (2012)** tried to initialize a RNN using all the tricks developed for ESN. He has shown that if the weights are initialized using the ESN methods, RNNs can be trained very effectively.
* It could learn quite well just by *learning* the *hidden-output connections*.
* However, it could learn even better if it also learn to make the *hidden-hidden weights* better. So **Ilya** tried using the *ESN's initializations* but then *training* with *back propagation through time (BPTT)*.
* He used ***rmsprop*** *with* ***momentum*** and he discovered that, that is actually a very effective way to train RNN.