January 31st 2005

Introduction to Neural Networks and Data Mining – lecture 3
BARGIELA Andrzej *(ISM – NTU, Hirota Lab. – TITech)*

13. Classification

13.1. Discriminants

Neural networks can also be used to classify data. Unlike regression problems, where the goal is to produce a particular output value for a given input, classification problems require us to label each data point as belonging to one of *n* classes. Neural networks can do this by learning a **discriminant** function, which **separates** the classes. For example, a network with a single linear output can solve a two-class problem by learning a discriminant function which is greater than zero for one class, and less than zero for the other. Fig. 6 shows two such two-class problems, with filled dots belonging to one class, and unfilled dots to the other. In each case, a line is drawn where a discriminant function that separates the two classes is zero.

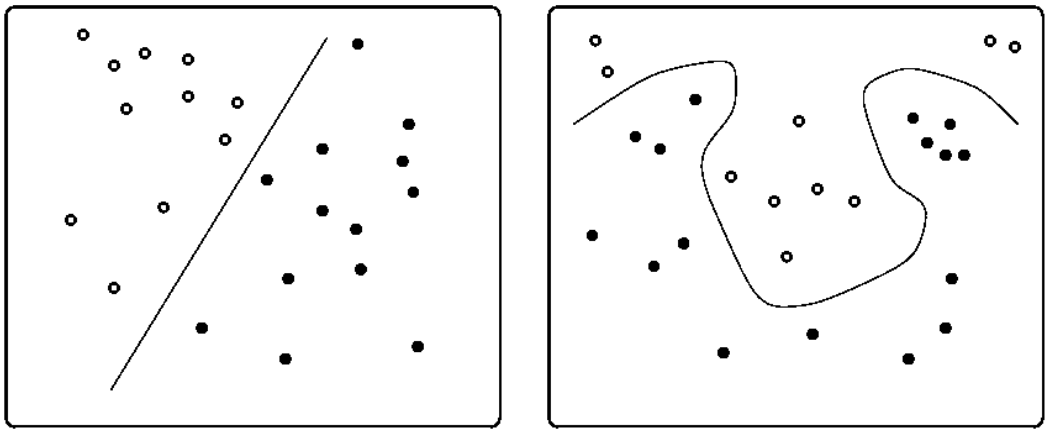


Figure 6.

On the left side, a straight line can serve as a discriminant: we can place the line such that all filled dots lie on one side, and all unfilled ones lie on the other. The classes are said to be **linearly separable**. Such problems can be learned by neural networks that have no hidden units. On the right side, a highly non-linear function is required to ensure class separation. This problem can be solved only by a neural network with hidden units.

13.2. Binomial

To use a neural network for classification, we need to construct an equivalent function approximation problem by assigning a target value for each class. For a **binomial** (two-class) problem we can use a network with a single output y, and binary target values: 1 for one class, and 0 for the other. We can thus interpret the network's output as an estimate of the probability that a given pattern belongs to the '1' class. To classify a new pattern after training, we then employ the **maximum likelihood** discriminant, y > 0.5.

A network with linear output used in this fashion, however, will expend a lot of its effort on getting the target values *exactly* right for its training points - when all we actually care about is the correct positioning of the discriminant. The solution is to use an activation function at the output that saturates at the two target values: such a function will be close to the target value for any net input that is sufficiently large and has the correct sign. Specifically, we use the logistic sigmoid function

$$f(u) = \frac{1}{1+e^{-u}} \qquad f'(u) = f(u)\,(1-f(u))$$

Given the probabilistic interpretation, a network output of, say, 0.01 for a pattern that is actually in the '1' class is a *much* more serious error than, say, 0.1. Unfortunately the sum-squared loss function makes almost no distinction between these two cases. A loss function that is appropriate for dealing with probabilities is the **cross-entropy** error. For the two-class case, it is given by

$$E = -t \ln y - (1-t) \ln(1-y)$$

When logistic output units and cross-entropy error are used together in backpropagation learning, the error signal for the output unit becomes just the difference between target and output:

$$\frac{\partial E}{\partial net} = \ldots = y - t$$

In other words, implementing cross-entropy error for this case amounts to nothing more than omitting the f'(net) factor that the error signal would otherwise get multiplied by. This is not an accident, but indicative of a deeper mathematical connection: cross-entropy error and logistic outputs are the "correct" combination to use for binomial probabilities, just like linear outputs and sum-squared error are for scalar values.

13.3. Multinomial

If we have multiple *independent* binary attributes by which to classify the data, we can use a network with multiple logistic outputs and cross-entropy error. For **multinomial** classification problems (1-of-n, where n > 2) we use a network with n outputs, one

corresponding to each class, and target values of 1 for the correct class, and 0 otherwise. Since these targets are not independent of each other, however, it is no longer appropriate to use logistic output units. The corect generalization of the logistic sigmoid to the multinomial case is the **softmax** activation function:

$$f(net_i) = \frac{e^{net_i}}{\sum_o e^{net_o}}$$

where o ranges over the n output units. The cross-entropy error for such an output layer is given by

$$E = -\sum_o t_o \ln y_o$$

Since all the nodes in a softmax output layer interact (the value of each node depends on the values of all the others), the derivative of the cross-entropy error is difficult to calculate. Fortunately, it again simplifies to

$$\frac{\partial E}{\partial net} = \ldots = y - t$$

so we don't have to worry about it.

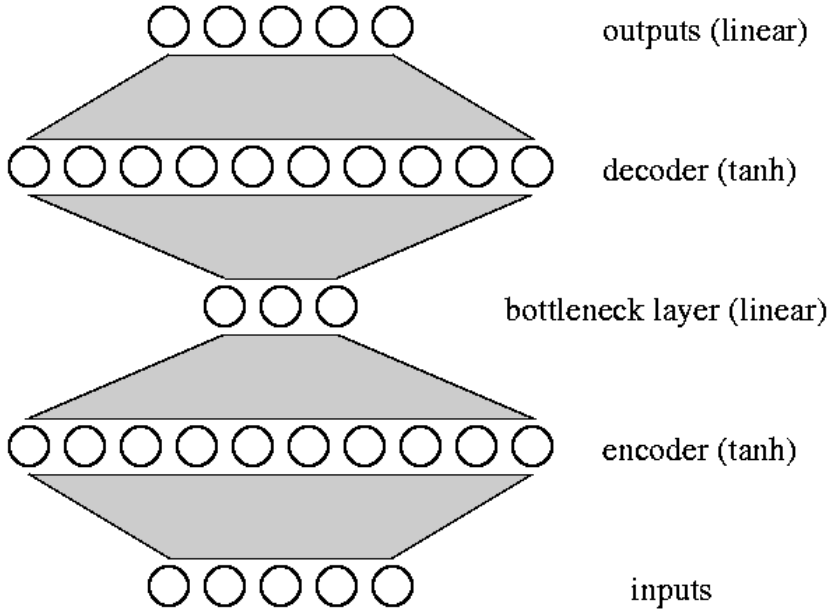
14. Non-supervised learning

It is possible to use neural networks to learn about data that contains neither target outputs nor class labels. There are many tricks for getting error signals in such **non-supervised** settings; here we'll briefly discuss a few of the most common approaches: autoassociation, time series prediction, and reinforcement learning.

14.1. Autoassociation

Autoassociation is based on a simple idea: if you have inputs but no targets, just use the inputs as targets. An autoassociator network thus tries to learn the identity function. This is only non-trivial if the hidden layer forms an **information bottleneck** - contains less units than the input (output) layer, so that the network must perform **dimensionality reduction** (a form of data compression).

A linear autoassociator trained with sum-squared error in effect performs **principal component analysis** (PCA), a well-known statistical technique. PCA extracts the subspace (directions) of highest variance from the data. As was the case with regression, the linear neural network offers no direct advantage over known statistical methods, but it does suggest an interesting nonlinear generalization:



Figure

This **nonlinear autoassociator** includes a hidden layer in both the encoder and the decoder part of the network. Together with the linear bottleneck layer, this gives a network with at least 3 hidden layers. Such a deep network should be preconditioned if it is to learn successfully.

14.2. Time series prediction

When the input data x forms a temporal series, an important task is to predict the next point: the weather tomorrow, the stock market 5 minutes from now, and so on. We can (attempt to) do this with a feedforward network by using **time-delay embedding**: at time t, we give the network x(t), x(t-1), ... x(t-d) as input, and try to predict x(t+1) at the output. After propagating activity forward to make the prediction, we wait for the actual value of x(t+1) to come in before calculating and backpropagating the error. Like all neural network architecture parameters, the dimension d of the embedding is an important but difficult choice.

A more powerful (but also more complicated) way to model a time series is to use recurrent neural networks.

14.3. Reinforcement learning

Sometimes we are faced with the problem of **delayed reward**: rather than being told the correct answer for each input pattern immediately, we may only occasionally get a positive or negative reinforcement signal to tell us whether the entire sequence of

actions leading up to this was good or bad. Reinforcement learning provides ways to get a continuous error signal in such situations.

Q-learning associates an expected utility (the Q-value) with each action possible in a particular state. If at time t we are in state s(t) and decide to perform action a(t), the corresponding Q-value is updated as follows:

$$Q(s(t), a(t)) = r(t) + \gamma \max_a Q(s(t+1), a)$$

where r(t) is the **instantaneous reward** resulting from our action, s(t+1) is the state that it led to, a are all possible actions in that state, and gamma <= 1 is a **discount factor** that leads us to prefer instantaneous over delayed rewards.

A common way to implement Q-learning for small problems is to maintain a table of Q-values for all possible state/action pairs. For large problems, however, it is often impossible to keep such a large table in memory, let alone learn its entries in reasonable time. In such cases a neural network can provide a compact approximation of the Q-value function. Such a network takes the state s(t) as its input, and has an output y_a for each possible action. To learn the Q-value Q(s(t), a(t)), it uses the right-hand side of the above Q-iteration as a target:

$$\delta_{a(t)} = r(t) + \gamma \max_a y_a - y_{a(t)}$$

Note that since we require the network's outputs at time t+1 in order to calculate its error signal at time t, we must keep a one-step memory of all input and hidden node activity, as well as the most recent action. The error signal is applied only to the output corresponding to that action; all other output nodes receive no error (they are "don't cares").

TD-learning is a variation that assigns utility values to states alone rather than state/action pairs. This means that search must be used to determine the value of the best successor state. TD(λ) replaces the one-step memory with an exponential average of the network's gradient; this is similar to momentum, and can help speed the transport of delayed reward signals across large temporal distances.

One of the most successful applications of neural networks is **TD-Gammon**, a network that used TD(λ) to learn the game of backgammon from scratch, by playing only against itself. TD-Gammon is now the world's strongest backgammon program, and plays at the level of human grandmasters.

15. Recurrent Networks

Consider the following two networks:

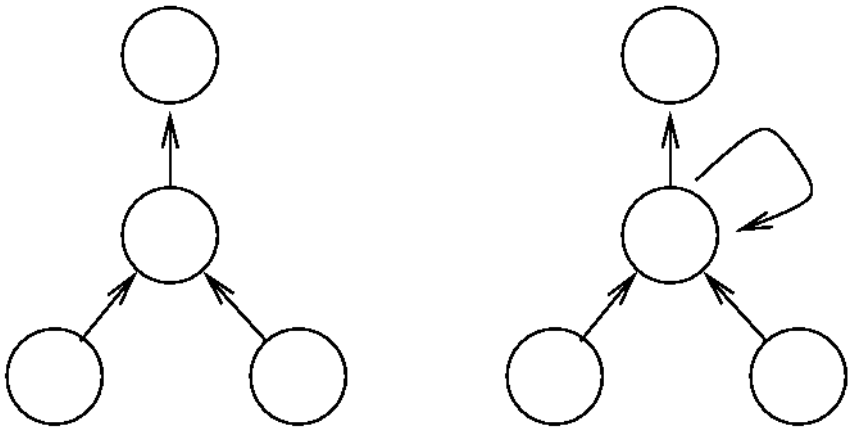


Figure. 1

The network on the left is a simple feed forward network of the kind we have already met. The right hand network has an additional connection from the hidden unit *to itself*. What difference could this little weight make?

Each time a pattern is presented, the unit computes its activation just as in a feed forward network. However its net input now contains a term, which reflects the state of the network (the hidden unit activation) before the pattern was seen. When we present subsequent patterns, the hidden and output units' states will be a function of everything the network has seen so far. The network has a sense of history, and we must think of pattern presentation as it happens in time.

15.1. Network topology

Once we allow feedback connections, our network topology becomes very free: we can connect any unit to any other, even to itself. Two of our basic requirements for computing activations and errors in the network are now violated. When computing activations, we required that before computing y_i, we had to know the activations of all units in the anterior set, A_i. For computing errors, we required that before computing δ_j, we had to know the errors of all units in its posterior set P_j.

For an arbitrary unit in a recurrent network, we now define its activation *at time t* as:

$$y_i(t) = f_i(net_i(t\text{-}1))$$

At each time step, therefore, activation propagates forward through one layer of connections only. Once some level of activation is present in the network, it will continue to flow around the units, even in the absence of any new input whatsoever. We can now present the network with a time series of inputs, and require that it produce an output based on this series. This presents a whole set of new problems, which can be addressed by the networks, as well as some rather difficult matters concerning training.

Before we address the new issues in training and operation of recurrent neural networks, let us first look at some sample tasks, which have been attempted (or solved) by such networks.

- Speech recognition

  In some of the best speech recognition systems built so far, speech is first presented as a series of spectral slices to a recurrent network. Each output of the network represents the probability of a specific phonem (speech sound, e.g. /i/, /p/, etc), given both present and recent input. The probabilities are then interpreted by a Hidden Markov Model, which tries to recognize the whole utterance.

- Music composition

  A recurrent network can be trained by presenting it with the notes of a musical score. It's task is to predict the next note. Obviously this is impossible to do perfectly, but the network learns that some notes are more likely to occur in one context than another. Training, for example, on a lot of music by J. S. Bach, we can then seed the network with a musical phrase, let it predict the next note, feed this back in as input, and repeat, generating new music. Music generated in this fashion typically sounds fairly convincing at a very local scale, i.e. within a short phrase. At a larger scale, however, the compositions wander randomly from key to key, and no global coherence arises. This is an interesting area for further work....

15.2. The simple recurrent network

One way to meet these requirements is illustrated below in a network known variously as an Elman network (after Jeff Elman, the originator), or as a Simple Recurrent Network. At each time step, a copy of the hidden layer units is made to a copy layer. Processing is done as follows:

1. Copy inputs for time *t* to the input units
2. Compute hidden unit activations using net input from input units and from copy layer
3. Copy new hidden unit activations to copy layer
4. Compute output unit activations as usual

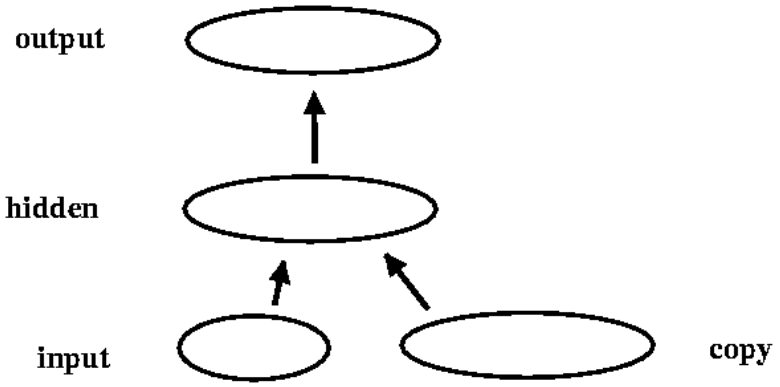


Figure 2.

In computing the activation, we have eliminated cycles, and so our requirement that the activations of all posterior nodes be known is met. Likewise, in computing errors, all trainable weights are feed forward only, so we can apply the standard backpropagation algorithm as before. The weights from the copy layer to the hidden layer play a special role in error computation. The error signal they receive comes from the hidden units, and so depends on the error at the hidden units at time *t*. The activations in the hidden units, however, are just the activation of the hidden units at time *t-1*. Thus, in training, we are considering a gradient of an error function, which is determined by the activations at the present and the previous time steps.

A generalization of this approach is to copy the input and hidden unit activations for a number of previous timesteps. The more context (copy layers) we maintain, the more history we are explicitly including in our gradient computation. This approach has become known as Back Propagation Through Time. It can be seen as an approximation to the ideal of computing a gradient which takes into consideration not just the most recent inputs, but all inputs seen so far by the network. The figure below illustrates one version of the process:

The inputs and hidden unit activations at the last three time steps are stored. The solid arrows show how each set of activations is determined from the input and hidden unit activations on the previous time step. A backward pass, illustrated by the dashed arrows, is performed to determine separate values of delta (the error of a unit with respect to its net input) for each unit **and** each time step separately. Because each earlier layer is a copy of the layer one level up, we introduce the new constraint that the weights at each level be identical. Then the partial derivative of the negative error with respect to $w_{i,j}$ is simply the sum of the partials calculated for the copy of $w_{i,j}$ between each two layers.

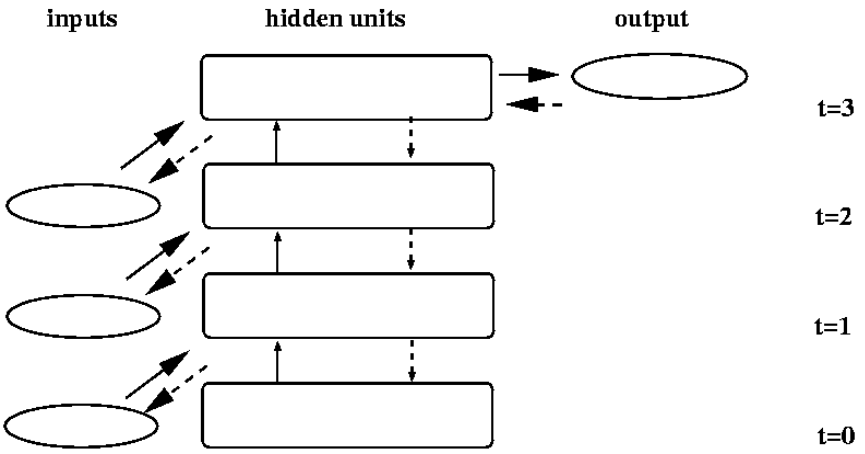


Figure 3.

Elman networks and their generalization, Back Propagation Through Time, both seek to approximate the computation of a gradient based on all past inputs, while retaining the standard back prop algorithm. In the next section we will see how we can compute the true temporal gradient using a method known as Real Time Recurrent Learning.

16. Real Time Recurrent Learning

In deriving a gradient-based update rule for recurrent networks, we now make network connectivity very unconstrained. We simply suppose that we have a set of input units, $I = \{x_k(t),\ 0<k<m\}$, and a set of other units, $U = \{y_k(t),\ 0<k<n\}$, which can be hidden or output units. To index an arbitrary unit in the network we can use

$$z_k(t) = \begin{cases} x_k(t) & \text{if } k \in I \\ y_k(t) & \text{if } k \in U \end{cases} \qquad (1)$$

Let **W** be the weight matrix with n rows and $n+m$ columns, where $w_{i,j}$ is the weight to unit i (which is in U) from unit j (which is in I or U). Units compute their activations in the now familiar way, by first computing the weighted sum of their inputs:

$$\text{net}_k(t) = \sum_{l \in U \cup I} w_{kl} z_l(t) \qquad (2)$$

where the only new element in the formula is the introduction of the temporal index t. Units then compute some non-linear function of their net input

$$y_k(t+1) = f_k(\text{net}_k(t)) \qquad (3)$$

Usually, both hidden and output units will have non-linear activation functions. Note that external input at time *t* does not influence the output of any unit until time *t+1*. The network is thus a discrete dynamical system.

Some of the units in *U* are output units, for which a target is defined. A target may not be defined for every single input however. For example, if we are presenting a string to the network to be classified as either grammatical or ungrammatical, we may provide a target only for the last symbol in the string. In defining an error over the outputs, therefore, we need to make the error time dependent too, so that it can be undefined (or 0) for an output unit for which no target exists at present. Let *T(t)* be the set of indices *k* in *U* for which there exists a target value $d_k(t)$ at time *t*. We are forced to use the notation d_k instead of *t* here, as *t* now refers to time. Let the error at the output units be

$$e_k(t) = \begin{cases} d_k(t) - y_k(t) & \text{if } k \in T(t) \\ 0 & \text{otherwise} \end{cases} \qquad (4)$$

and define our error function for a single time step as

$$E(\tau) = \frac{1}{2} \sum_{k \in U} [e_k(\tau)]^2 \qquad (5)$$

The error function we wish to minimize is the sum of this error over all past steps of the network

$$E_{\text{total}}(t_0, t_1) = \sum_{\tau = t0+1}^{t1} E(\tau) \qquad (6)$$

Now, because the total error is the sum of all previous errors and the error at this time step, so also, the gradient of the total error is the sum of the gradient for this time step and the gradient for previous steps

$$\nabla_{\mathbf{W}} E_{total}(t_0, t+1) = \nabla_{\mathbf{W}} E_{total}(t_0, t) + \nabla_{\mathbf{W}} E(t+1) \qquad (7)$$

As a time series is presented to the network, we can accumulate the values of the gradient, or equivalently, of the weight changes. We thus keep track of the value

$$\Delta w_{ij}(t) = -\mu \frac{\partial E(t)}{\partial w_{ij}} \qquad (8)$$

After the network has been presented with the whole series, we alter each weight w_{ij} by

$$\sum_{t=t0+1}^{t1} \Delta w_{ij}(t) \quad (9)$$

We therefore need an algorithm that computes

$$-\frac{\partial E(t)}{\partial w_{ij}} = -\sum_{k \in U} \frac{\partial E(t)}{\partial y_k(t)} \frac{\partial y_k(t)}{\partial w_{ij}} = \sum_{k \in U} e_k(t) \frac{\partial y_k(t)}{\partial w_{ij}} \quad (10)$$

at each time step t. Since we know $e_k(t)$ at all times (the difference between our targets and outputs), we only need to find a way to compute the second factor $\partial y_k(t) / \partial w_{ij}$.

IMPORTANT

The key to understanding RTRL is to appreciate what this factor expresses. It is essentially a measure of the sensitivity of the value of the output of unit k at time t to a small change in the value of w_{ij}, taking into account the effect of such a change in the weight over the entire network trajectory from t_0 to t. Note that w_{ij} does not have to be connected to unit k. Thus this algorithm is non-local, in that we need to consider the effect of a change at one place in the network on the values computed at an entirely different place. Make sure you understand this before you dive into the derivation given next

16.1. Derivation of time-dependent sensitivity of outputs

This is given here for completeness, for those who wish perhaps to implement RTRL. Make sure you at least know what role the factor $\partial y_k(t) / \partial w_{ij}$ plays in computing the gradient.

From Equations 2 and 3, we get

$$\frac{\partial y_k(t+1)}{\partial w_{ij}} = f_k'(\mathrm{net}_k(t)) \left[\sum_{l \in U \cup I} w_{kl} \frac{\partial z_l(t)}{\partial w_{ij}} + \delta_{ik} z_j(t) \right] \quad (11)$$

where δ_{ik} is the Kronecker delta

$$\delta_{ik} = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise} \end{cases} \quad (12)$$

[Exercise: Derive Equation 11 from Equations 2 and 3]

Because input signals do not depend on the weights in the network,

$$\frac{\partial z_l(t)}{\partial w_{ij}} = 0 \text{ for } l \in I \quad (13)$$

Equation 11 becomes:

$$\frac{\partial y_k(t+1)}{\partial w_{ij}} = f_k'(\text{net}_k(t)) \left[\sum_{l \in U} w_{kl} \frac{\partial y_l(t)}{\partial w_{ij}} + \delta_{ik} z_j(t) \right] \quad (14)$$

This is a recursive equation. That is, if we know the value of the left hand side for time 0, we can compute the value for time 1, and use that value to compute the value at time 2, etc. Because we assume that our starting state ($t = 0$) is independent of the weights, we have

$$\frac{\partial y_k(t_0)}{\partial w_{ij}} = 0 \quad (15)$$

These equations hold for all $k \in U, i \in U \text{ and } j \in U \cup I$.

We therefore need to define the values

$$p_{ij}^k(t) = \frac{\partial y_k(t)}{\partial w_{ij}} \quad (16)$$

for every time step t and all appropriate i, j and k. We start with the initial condition

$$p_{ij}^k(t_0) = 0 \quad (17)$$

and compute at each time step

$$p_{ij}^{k}(t+1) = f_k'(\mathrm{net}_k(t)) \left[\sum_{l \in U} w_{kl} p_{ij}^{l}(t) + \delta_{ik} z_j(t) \right] \qquad (18)$$

The algorithm then consists of computing, at each time step t, the quantities $p_{ij}^{k}(t)$ using equations 17 and 18, and then using the differences between targets and actual outputs to compute weight changes

$$\Delta w_{ij}(t) = \mu \sum_{k \in U} e_k(t) p_{ij}^{k}(t) \qquad (19)$$

and the overall correction to be applied to w_{ij} is given by

$$\Delta w_{ij} = \sum_{t=t0+1}^{t1} \Delta w_{ij}(t) \qquad (20)$$

17. Dynamics of RNNs

Consider the recurrent network illustrated below. A single input unit is connected to each of the three "hidden" units. Each hidden unit in turn is connected to itself and the other hidden units. As in the RTRL derivation, we do not distinguish now between hidden and output units. Any activation which enters the network through the input node can flow around from one unit to another, potentially forever. Weights less than 1.0 will exponentially reduce the activation, weights larger than 1.0 will cause it to increase. The non-linear activation functions of the hidden units will hopefully prevent it from growing without bound.

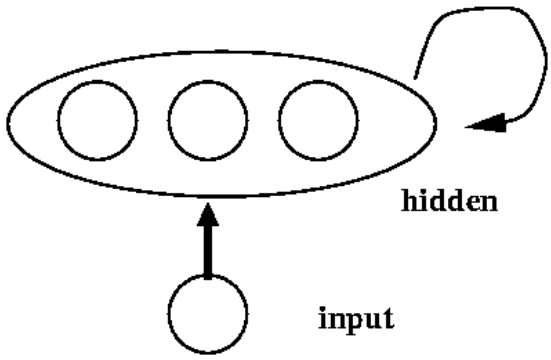


Figure 1.

As we have three hidden units, their activation at any given time *t* describes a point in a 3-dimensional state space. We can visualize the temporal evolution of the network state by watching the state evolve over time.

In the absence of input, or in the presence of a steady-state input, a network will usually approach a fixed point attractor. Other behaviors are possible, however. Networks can be trained to oscillate in regular fashion, and chaotic behavior has also been observed. The development of architectures and algorithms to generate specific forms of dynamic behavior is still an active research area.

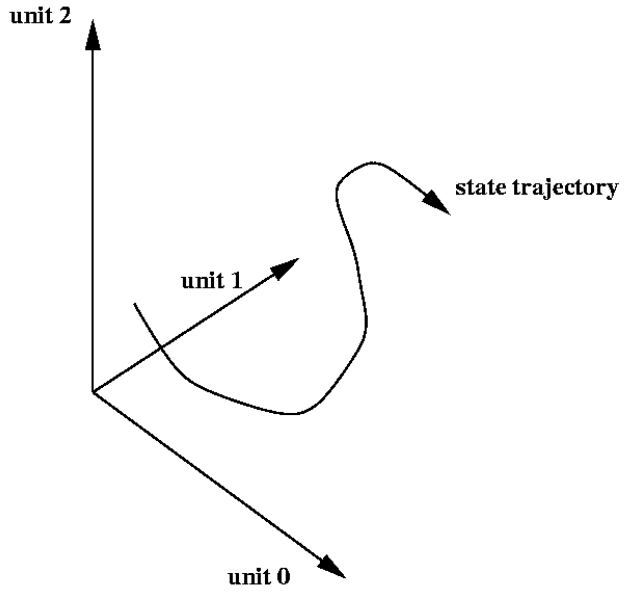


Figure. Xx

17.1. Some limitations of gradient methods and RNNs.

The simple recurrent network computed a gradient based on the present state of the network and its state one time step ago. Using Back Prop Through Time, we could compute a gradient based on some finite *n* time steps of network operation. RTRL provided a way of computing the true gradient based on the complete network history from time 0 to the present. Is this perfection?

Unfortunately not. With feedforward networks, which have a large number of layers, the weights which are closest to the output are the easiest to train. This is no surprise, as their contribution to the network error is direct and easily measurable. Every time we back propagate an error one layer further back, however, our estimate of the contribution of a particular weight to the observed error becomes more indirect. You can think of error flowing in the top of the network in distinct streams. Each back propagation dilutes the error, mixing up error from distinct sources, until, far back in the network, it becomes virtually impossible to tell who is responsible for what. The error signal has become completely diluted.

With RTRL and BPTT we face a similar problem. Error is now propagated back in time, but each time step is exactly equivalent to propagating through an additional layer of a feed forward network. The result, of course, is that it becomes very difficult to assess the importance of the network state at times, which lie far back in the past. Typically, gradient based networks cannot reliably use information which lies more than about 10 time steps in the past. If you now imagine an attempt to use a recurrent neural network in a real life situation, e.g. monitoring an industrial process, where data are presented as a time series at some realistic sampling rate (say 100 Hz), it becomes clear that these networks are of limited use. The next section shows a recent model, which tries to address this problem.

18. Long Short-Term Memory

In a recurrent network, information is stored in two distinct ways. The activations of the units are a function of the recent history of the model, and so form a short-term memory. The weights too form a memory, as they are modified based on experience, but the timescale of the weight change is much slower than that of the activations. We call those "a long-term memory". The Long Short-Term Memory model is an attempt to allow the unit activations to retain important information over a much longer period of time than the 10 to 12 time steps, which is the limit of RTRL or BPTT models.

The figure below shows a maximally simple LSTM network, with a single input, a single output, and a single memory block in place of the familiar hidden unit.

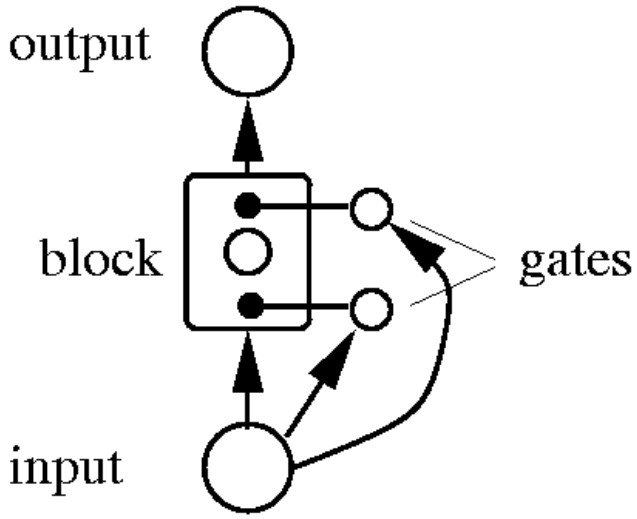


Figure xx

This figure above shows a maximally simple LSTM network, with a single input, a single output, and a single memory block in place of the familiar hidden unit. Each block has two associated gate units (details below). Each layer may, of course, have multiple units or blocks. In a typical configuration, the first layer of weights is provided from input to the blocks and gates. There are then recurrent connections from one block

to other blocks and gates. Finally there are weights from the blocks to the outputs. The next figure shows the details of the memory block in more detail.

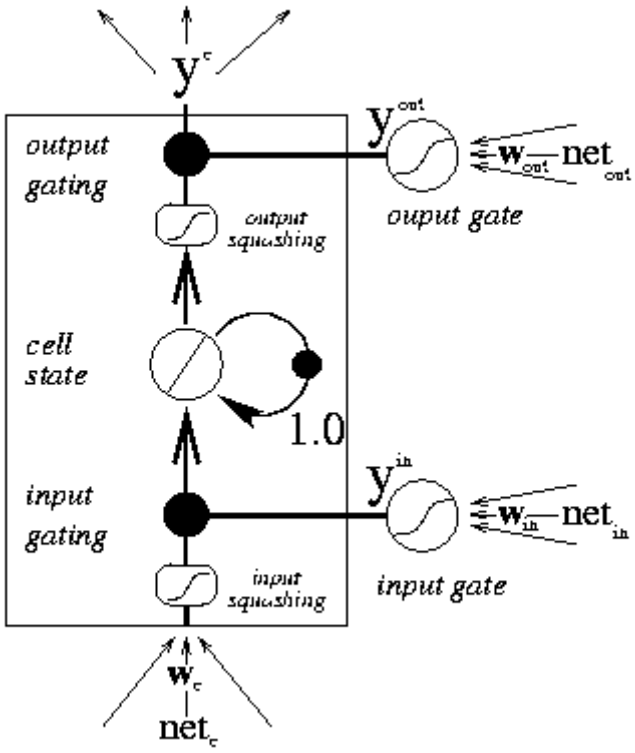


Figure xx

The hidden units of a conventional recurrent neural network have now been replaced by memory **blocks**, each of which contains one or more memory **cells**. At the heart of the cell is a simple linear unit with a single self-recurrent connection with weight set to 1.0. In the absence of any other input, this connection serves to preserve the cell's current state from one moment to the next. In addition to the self-recurrent connection, cells receive input from input units and other cell and gates. While the cells are responsible for maintaining information over long periods of time, the responsibility for deciding what information to store, and when to apply that information lies with an input and output gating unit, respectively.

The input to the cell is passed through a non-linear squashing function (*g(x)*, typically the logistic function, scaled to lie within [-2,2]), and the result is then multiplied by the output of the input gating unit. The activation of the gate ranges over [0,1], so if its activation is near zero, nothing can enter the cell. Only if the input gate is sufficiently active is the signal allowed in. Similarly, nothing emerges from the cell unless the output gate is active. As the internal cell state is maintained in a linear unit, its activation range is unbounded, and so the cell output is again squashed when it is released (*h(x)*, typical range [-1,1]). The gates themselves are nothing more than

conventional units with sigmoidal activation functions ranging over [0,1], and they each receive input from the network input units and from other cells.

Thus we have:

- Cell output: $y^c_j(t)$ is

  $y^c_j(t) = y^{out}_j(t)\ h(s_{cj}(t))$

- where $y^{out}_j(t)$ is the activation of the output gate, and the state, $s_{cj}(t)$ is given by

  $s_{cj}(0) = 0$, and

  $s_{cj}(t) = s_{cj}(t\text{-}1) + y^{in}_j(t)\ g(\text{net}_{cj}(t))$ for $t > 0$.

This division of responsibility---the input gates decide what to store, the cell stores information, and the output gate decides when that information is to be applied---has the effect that salient events can be remembered over arbitrarily long periods of time. Equipped with several such memory blocks, the network can effectively attend to events at multiple time scales.

Network training uses a combination of RTRL and BPTT, and we won't go into the details here. However, consider an error signal being passed back from the output unit. If it is allowed into the cell (as determined by the activation of the output gate), it is now trapped, and it gets passed back through the self-recurrent connection indefinitely. It can only affect the incoming weights, however, if it is allowed to pass by the input gate.

On selected problems, an LSTM network can retain information over arbitrarily long periods of time; over 1000 time steps in some cases. This gives it a significant advantage over RTRL and BPTT networks on many problems. In particular LSTM have been used to distinguish between different spoken languages based on speech prosody (roughly: the melody and rhythm of speech).

Acknowledgements

Prof. Bargiela would like to acknowledge JSPS funding to deliver a series of lectures on *Neural Networks and Data Mining* for postgraduate students at the Tokyo Institute of Technology. The lectures have been prepared using the web material produced by Nicolas Schraudolph and Fred Cummins from IDISIA, Switzerland and the book by David MacKay entitled *Information Theory, Inference and Learning Algorithms* published by Cambridge University Press.

Bibliography

Schraudolph and Graepel**,** Towards Stochastic Conjugate Gradient Methods, *Proc. 9th Intl. Conf. Neural Information Processing*, Singapore. IEEE, 2002

Graepel and Schraudolph, Stable Adaptive Momentum for Rapid Online Learning in Nonlinear Systems, *Proc. Intl. Conf. Artificial Neural Networks*, Madrid. Springer Verlag, Berlin 2002

Schraudolph and Giannakopoulos, Online Independent Component Analysis With Local Learning Rate Adaptation, *Advances in Neural Information Processing Systems 12*, MIT Press, Cambridge 2000

Schraudolph**,** Slope Centering: Making Shortcut Weights Effective, *Proc. 8th International Conference on Artificial Neural Networks*, Skövde. Springer Verlag, Berlin 1998

Schraudolph, Centering Neural Network Gradient Factors, in: Orr and Müller (eds.), *Neural Networks: Tricks of the Trade*, Springer Verlag, Berlin 1998

Schraudolph and Sejnowski, Tempering Backpropagation Networks: Not All Weights are Created Equal, *Advances in Neural Information Processing Systems 8*, MIT Press, Cambridge 1996

Schraudolph, Dayan, and Sejnowski, Learning to Evaluate Go Positions via Temporal Difference Methods, in: Baba and Jain (eds.), *Computational Intelligence in Games*, Springer Verlag, Berlin 2001

Schmidhuber, Zhao, and Schraudolph, Reinforcement Learning with Self-Modifying Policies, in: *Learning to Learn*, Kluwer Academic Publishers, Norwell 1998

Schraudolph, Eldracher, and Schmidhuber, Processing Images by Semi-Linear Predictability Minimization, *Network: Computation in Neural Systems* 10(2), 1999

Cummins, F., Prosodic characteristics of synchronous speech, in Puppel, S. and Demenko, G., editors, Prosody 2000: Speech Recognition and Synthesis, pages 45--49, Krakow, Poland. Adam Mickiewicz University, 2000.

Gers, F. A., Schmidhuber, J., and Cummins, F., Learning to forget: Continual prediction with LSTM, Neural Computation 12 (10):2451—2471, 2000.

Cummins, F., Doherty, C., and Dilly, L., Discrimination of pitch change in speech- and non-speech stimuli, In Proceedings of the 15th Artificial Intelligence and Cognitive Science Conference, pages 29-38, Castlebar, Ireland, 2004.

David MacKay, Information Theory, Inference and Learning Algorithms, Cambridge University Press, 2005