Brain Learning Algorithms, Meeting 1

Nancy Lynch

February 14, 2020

Bring:
Summary of today's meeting, and tentative plans for the semester, from final reminder email.
These notes.

1 Context, background, and goals

We are interested in algorithms for biologically-plausible neural networks. This means real theory, including formal modeling, algorithms, verification, analysis, lower bounds, impossibility results, and so on.

In Spring, 2019, we held a reading group that covered:

1. Basic models of biologically-plausible neural networks, mainly our own model of discrete, synchronous, usually-stochastic Spiking Neural Network model, also a rate-based model.
2. Problems of focus and attention: WTA, k-WTA
3. Problems of neural coding/representation: Neuro-RAM, dimensionality reduction
4. Learning, just started at the end, with a brief study of Hebbian rules, Oja,...

This term, I would like to focus on learning. This means learning in *biologically-plausible neural networks*. We emphasize that this is not the same as learning in typical Artificial Neural Networks (ANNs): Rather, we want important biological restrictions such as limited network size and speed, and locality of action.

I think it may be useful to think about gradient descent and the like as it seems appropriate this semester. But when we do, we should try to figure out how to make these strategies work in biological networks. Can we capture key ideas from ANNs within our model, or some reasonable extensions of our model?

Kinds of learning to consider: We will consider *both supervised and unsupervised learning*, Probably mostly unsupervised.

I am particularly interested in learning of *structured concepts.* A thesis I am adopting is that real learning makes heavy use of structure in the input data space.

Examples of kinds of structure include:

1. *Logical structure:* Hierarchy; logical connectives such as ands, ors, nots, implications,...; associations; linguistic structure.
2. *Real-world structure:* Spatial, geometric, graph-theoretical as an abstraction, temporal.

1

I hope that this reading group can lead to some new research.

Recommended background reading and listening: To prepare for this group, I read the following:

1. Valiant's book, and a few of his papers. These are mostly older but still very interesting.
2. Many papers by Papadimitrou and his co-workers.
3. Rojas's book on neural networks, skimmed/read most of it, but esp. Chapters 5, 13, 14, and 15.
4. Oja's papers: Principal Components, Minor Components, and Linear Neural Networks, and Neural Networks, Principal Components, and Subspaces.

It is also worth looking at some on-line courses:

1. Nancy Kanwisher's course on "The Human Brain".
   https://nancysbraintalks.mit.edu/
   https://nancysbraintalks.mit.edu/video/nancys-ted-talk-neural-portrait-human-mind
2. Christos Papadimitriou's on-line course on "Computation and the Brain".
   https://computationandbrain.github.io/about/
   Click on "Resources" on the top right for papers, lectures, etc.
3. Sanjoy Dasgupta's course on "Neurally-inspired unsupervised learning":
   http://cseweb.ucsd.edu/ dasgupta/254-neural-ul/index.html

2 Kinds of brain models I'm (mainly) interested in

I'm relying here on our series of recent papers:

1. Lynch, Musco, Parter. WTA paper, which uses a basic model. New ArXiv version or Cameron's thesis are the best versions.
2. Su, Chang, Lynch k-WTA paper, which uses an extended version of the model.
3. Lynch, Musco. Composition paper.
4. Lynch, Musco, Parter. Neuro-RAM paper.
5. Hitron, Parter; Wang, 2 counting network papers.
6. Hitron, Lynch, Musco, Parter. Clustering and short-term memory (the "renaming paper".
7. Lynch, Mallmann-Trenn. Learning of hierarchically-structured concepts.
8. Chou, Wang. Oja analysis.

[1]I note here that others, e.g., in LIDS, study structure in input data space, by which they generally mean something else: the inputs are vectors chosen from some particular kinds of probability distributions.

The models I'm interested in all fit into the category of "synchronous Spiking Neural Networks".

Spiking Neural Networks refer to models in which the neurons fire at discrete times, as contrasted with (more abstract) models in which the neurons have only associated real values such as "firing rates" or "strengths" or "potentials", which change continuously.

Synchronous means that they operate in discrete rounds. The use of synchrony is reasonable because brains (as well as other biological systems) do have strong notions of time. Of course, the assumption of perfectly synchronized rounds is unrealistically strong, so we should consider some variations; for example, we can consider various forms of noise in the system.

If we regard the times as very fine-grained, we will have plenty of flexibility in specifying interesting spike timings.

We usually consider *stochastic activation*, using some type of sigmoid function to convert potential to firing probability. However, sometimes it is enough to restrict to simple deterministic threshold gates. These are less biologically plausible, but may be simpler to analyze because they avoid some of the probability. (But we might still have to deal with probabilistic inputs.)

Stochastic activation captures some aspects of noise. Also allows us to design strategies that use randomness, such as symmetry-breaking strategies and random sampling methods.

Types of information in the state:

1. Firing status for the latest round. This is all we used in our basic model, in the initial work by Lynch, Musco, Parter.

2. Some other components, which were added to the model as we found we needed them. Specifically, Lili generalized the simple state model to a more general model, in her k-WTA paper. The information her neurons maintain is a combination of its recent firing history, (up to some memory bound m), and its recent incoming (aggregated) potentials (again up to memory bound m).

3. Weight information for incoming edges. Frederik and I included weights of incoming edges in the state of a neuron. Seems like a convenient place to put them. Most other models seem to have separate model components for edge weights, essentially associating "state" with the edges as well as the neurons.

4. Other possibilities: Valiant includes the threshold in the state and allows it to be changed, based on a program from a rich class of programs. It is not clear whether we would want this. In fact, he allows arbitrary real-valued state components. Valiant's model is a very powerful model, emphasizing flexible programming.

Things we don't include:

1. We don't keep track of firing of individual incoming neighbors, just the aggregate incoming potential. So a neuron can't distinguish where the potential comes from, can just react to the aggregate. This seems to be consistent with biology.

   It's also like what Emily Toomey and Karl Berggren are doing in their hardware SNNs. In fact, they use a separate "inductor" component for each neuron just to do the aggregation. Rojas' book also describes this type of separation, in some of the models he discusses.

2. We don't try to abstract everything into a real-valued quantity such as a "firing rate". We do allow real-valued state components, in the form of aggregated potentials, in our extended models. These might be somewhat different: potentials seem to capture short-term effects, summarizing recent firing history, whereas rate seems to be a longer-term characteristic. But this is a matter of degree (how much memory we keep). In addition, we include discrete spiking information.

What we have done with these models: We have so far addressed problems of WTA, neuro-RAM, similarity detection, clustering, renaming, counting.

Math techniques: Automata-theoretic, discrete analysis, probability.

Networks for the above problems are all hand-designed, or evolved, not learned. Except that the Hitron et al. paper describes a kind of short-term memory. (Cameron will speak about this next week.)

However, we have just recently moved to consider actual learning, with weight changes, in particular, of hierarchically-structured concepts. New paper just submitted to COLT and put on ArXiv this week.

Other models: For contrast.

1. *Rate-based model:* This abstracts from the low-level firing spikes to consider just real-valued "firing rates", or "strengths". There is no discrete information in the state or elsewhere in the system—just continuous values. The firing rate of each neuron is calculated by a differential equation, based on firing rates of incoming neurons.

   This seems too abstract for our purposes—these models can't even describe discrete events such as decisions or recognition.

2. *Valiant:* Discrete model.
   Seems not as formally specified as ours, esp. w.r.t. the distributed aspects.
   Valiant's model is very powerful and flexible. It is programmable, allowing general local programs to determine firing, and changes to weights and thresholds.
   It is not particularly bio-plausible, but he does deal with issues of amount of neural connectivity in the cortex.

   He addresses some very interesting problems of memorization and association. We should spend some time on this, to understand what he is trying to capture. It can suggest new problems for us to study.

3. *Papadimitriou:* He studies various problems using a special case of Valiant's model, in particular, memorization, recall, and association. His work on association is described in terms of "cell assemblies".

   He doesn't completely specify the learning model. We should revisit this, interesting problems and mechanisms.

3 Planning for presentations

Plan who can present what, consider suggestions for different material.

4 Rojas

Source: Raul Rojas. Neural Networks: A Systematic Introduction, esp. Chapters 5, 13, 14, and 15.

This is a pretty comprehensive study of learning in neural networks, ca. 1990s. All learning here is by edge weight adaptation. Covers many topics, notable:

1. Perceptrons.

2. Hebbian learning, Oja's rule.

3. Hopfield networks.

4. Boltzmann machines.

5. Kohonen networks.

Also gradient descent, backprop.

I will summarize the book...

4.1 The Biological Paradigm

He writes about neural computation in natural and artificial neural networks.

Biological neural networks: He describes the structure of biological neurons. Abstracting from this, all the models he considers in his book have input and output channels and a cell body. Input channels have associated weights.

Describes how biological neurons transmit information, in terms of ions and membrane potentials. He mentions Hodgkin and Huxley's very detailed electrical circuit model for neurons. Abstracting from this, he considers models based on directed graphs with "units" at the vertices.

Cells process information by integrating incoming signals and by reacting to inhibition. They basically sum the incoming signals and compare them with an internal threshold.

He describes how information is stored, and modified, in synapses of biological neurons. Abstracting, in models, we get weights associated with the edges, and certain rules for modifying the weights.

Artificial Neural Network (ANN) models: Networks of *units*, connected by directed edges with associated *weights*. Typical ANN learning rule: Hebbian learning.
Roughly speaking, the weight of an edge between two units is increased when the two units fire "simultaneously" and is decreased when the firing states of the two units are "uncorrelated".

Each unit computes a function from some class of *primitive functions*. More precisely, each unit computes the dot product of its inputs and the weights of the incoming channels, i.e., the incoming potential, and then computes some primitive function of the result.

ANN models differ mainly in the class of primitive functions used, and in their assumptions about the interconnection pattern, and the timing of the transmission of information.

If the ANN is a DAG, then the function computed by the overall network is determined in an obvious way from the individual neuron functions, the connectivity, and the weights. Non-DAG networks introduce complications, and timing becomes a consideration.

He is interested in how an ANN can be used to compute, or at least approximate, some function. In typical formulations, ANNs are "universal function approximators", in that they can approximate any infinitely-differentiable function. To demonstrate this, he presents a simple ANN that produces the first few coefficients in a Taylor series for an arbitrary infinitely-differentiable function.

Note that this universality result is about expressive power. However, this book is mostly about how a network can *learn* to approximate functions, i.e., how the weights can be determined.

4.2 Threshold logic

General stuff about networks. In general, he considers networks with real-valued inputs and real-valued outputs. But sometimes he restricts to Boolean input and output.

He describes both acyclic ("feed-forward") and cyclic ("recurrent") networks. For acyclic networks, it's easy to define what function it computes, based on the individual units' functions.

But if there are cycles, it's harder: we have to specify timing aspects, e.g., specifying one unit of time delay at each computing unit.
And we have to deal with issues of synchronizing the arrival of inputs and different units, and define when we read out the output.

He models his computing units as consisting of of two parts, one that aggregates the inputs, and the second that computes a primitive function of the resulting real-valued argument.

McCulloch-Pitts networks: McCulloch-Pitts units are a special case, using only binary signals and $\{0, 1\}$-valued functions. They have excitatory and inhibitory edges. In the basic MP model, a single incoming inhibitory edge is enough to prevent a unit from producing a signal. If no inhibition occurs, they act as threshold gates (output a binary value, but based on whether the total arriving potential exceeds its threshold).

McCulloch-Pitts networks can synthesize any logical function: Just provide individual units to recognize all the acceptable combinations of inputs, then combine them with an "or" unit. Of course, this can require a lot of units.

There are many variations on the MP network model (weighted edges, different assumptions about inhibition, allowing channels to transmit real-valued signals instead of just binary). The basic MP networks are powerful enough to simulate these, generally at some cost in network size.

Initial general classification of neural network models:

1. Binary vs. real-valued communication.

2. *Weighted vs. unweighted edges.* This has impact on the kind of learning that can occur. With unweighted edges, one typically considers modifying thresholds and connectivity. With weighted edges, one typically considers modifying thresholds and weights, but not connectivity.

3. *Synchronous vs. asynchronous models.* By "asynchronous", he means that the times (for input arrivals and for units to compute) are selected stochastically, not arbitrarily as usual in distributed computing theory.

4. Models with and without state stored in the individual units:

Optimization: He describes how techniques from *harmonic analysis* can be used to automatically synthesize networks to compute particular real-valued functions, as linear combinations of n individual functions.
He does this by mimizing a (quadratic) error for a candidate linear expression (an approximation). Compute the derivative of the error w.r.t. each of the coefficients. Use that to give a matrix equation that determines the needed coefficients. (The Hadamard-Walsh transform can be used to solve the equations.)

The resulting solutions can be wired as MP networks, using weighted edges and only two primitive operations, addition and binary multiplication.

4.3 The Perceptron

Weighted version of McCulloch-Pitts networks. Defined by Rosenblatt, refined by Minsky and Papert.
Allows computationally powerful units, with some constraints on inputs. Threshold elements.

Typical question: What kinds of patterns can be recognized by perceptron networks? This isn't a learning question—just expressiveness for recognition.

Minsky and Papert showed that we can't recognize everything with single-output perceptron networks; notably, the connectivity predicate cannot be recognized.

Expressive power of single perceptron units: This has been widely studied. Basically, all a single perceptron unit can do is separate the input space into two half-spaces. It can't compute (for example) an XOR.

Rojas characterizes a separating hyperplane by its defining m-vector of weights. These are just the coefficients in a linear inequality of the form $w_1 x_1 + ... + w_m x_m \geq 0$.

He considers the duality of "input space", i.e., the space of input vectors, and "weight space", e.g., the space of weight vectors. A plane in the input space corresponds to a point in the weight (vector) space.

A perceptron output (planar separator) is good if it has a small value for its "error function". Here the error function is simple, just the number of wrong answers (among a given set of inputs). The goal is to try to reduce this error function with a learning algorithm.

Perceptron learning algorithm: The usual perceptron learning algorithm starts with a "random" weight vector, then tries to reduce the value of the error function. After each new input, the algorithm corrects the weight vector with a simple additive adjustment: just add/subtract the input vector, depending on whether the answer output by the network was correct or incorrect for the new input.

Supervised and unsupervised learning: Rojas classifies learning algorithms as supervised vs. unsupervised. Supervised involves getting feedback for examples. Unsupervised gets no such feedback; it just runs, strengthening those communication paths that are used.

Supervised learning can be divided into methods that use reinforcement vs. error correction. The difference, in his classification, is just that, for reinforcement, we learn only whether or not the network produced the desired result. For error-correcting, we also learn the magnitude of the error.

Perceptron learning algorithm is classified as "supervised", since it uses training, with "reinforcement", since it gets just yes/no answers about each training example.

4.4 Unsupervised Learning and Clustering Algorithms

Now switch to considering *unsupervised learning.* Most biological learning is unsupervised.

Rojas considers two kinds of unsupervised learning, which he calls *reinforcement learning* and *competitive learning.* In reinforcement learning, each input produces a "reinforcement" of the network that is supposed to enhance the reproduction of the desired output. Hebbian learning, and Oja's rule, are examples of this.

In competitive learning, the output units "compete" somehow for the right to provide the output associated with the input vector.

Note: It seems that the networks in this chapter are single-layer only.

Unsupervised competitive learning: Assume that the input (vectors) fall into clusters. We would like to map each input vector to some particular output unit. All the inputs in the same cluster are supposed to map to the same output unit. So if we have k clusters, they should correspond to k output units.
Algorithm: For each new input vector, find the closest-aligned weight vector, and adjust it slightly in the direction of the input vector (and normalize).

Different update rules could be used here for the weight vectors:

1. Add a constant times the input vector, where the constant decays over time.

2. Update proportional to the difference between the input and weight vectors (again has the effect of slowing the update rates over time).

3. Batch update.

Presumably the decreasing rate of update is to try to achieve convergence. Rojas analyzes convergence.

How does this algorithm yield the needed clustering? Unclear. Assuming we have some way of identifying the possible choices for weight vectors, and comparing the new input with these vectors, it will find the best-aligned weight vector for each input vector.
But then how do we go from that to causing a particular corresponding output unit to fire?

This is a centralized learning algorithm. How to make it more neural/distributed? Can the selection of the output use some neuro-implementable WTA mechanism?

Unsupervised reinforcement learning: Push the weight vector towards a (function of the?) input vector. No special output units are chosen as in competitive learning described above. But presumably, we still have a form of clustering because "nearby" input vector presentations should lead to nearby weight vectors.

Here, Rojas considers networks of "linear associators", which just compute the weighted input, i.e., the potential, rather than a binary firing state. For example, he considers networks that compute the first Principal Component of the input vector. That is, they produce a weight vector w that maximizes the average (taken over all the presented input vectors) of the quadratic scalar products.

Oja's algorithm is designed for networks of linear associators. It computes the first PC. It requires a stopping condition, e.g., a predetermined number of iterations. The heart of the algorithm is Oja's formula for updating the weights at a unit:
Oja's formula:

$$w(t) = w(t-1) + \eta z(t-1) \cdot (x(t-1) - z(t-1) \cdot w(t-1)),$$

where $z(t-1)$ is the dot product of $w(t-1)$ and $x(t-1)$.

Rojas claims convergence, and automatic normalization.
This algorithm also adapts to changing inputs.
Applications: Pattern recognition, image compression

4.5 Learning as gradient descent

This seems like standard gradient descent material, involving supervised learning with a training set.

Assume that the activation functions are differentiable. Specifically, use a sigmoid function $s(x)$ for the activation function, everywhere.

Goal is to find the best combination of weights so that the function computed by the network approximates a given function f as closely as possible. The function is given only through training examples, not explicitly. All the given examples include both input and output, and all are correct examples of computations of f.

Idea: During training, try to minimize the value of the error function in weight space, Uses a quadratic error expression. To try to minimize, use gradient descent, and do the minimum calculation using a backpropagation algorithm.

We describe the algorithm first for feed-forward networks, then consider what has to happen to extend to the recurrent case.

Feed-forward networks: *Idea for the learning algorithm:* Initialize the weights randomly. Then for each new example, compute the gradient (direction of steepest descent) of the error function, use it to make small corrections in the weights. This is supposed to lead to a local minimum, presumably, under certain assumptions about the function and the examples.

Compute the gradient using a recursive "backpropagation" method: First, extend the network so it computes the error function automatically. Then the problem reduces to that of calculating the gradient of a new network function (the error) w.r.t. the weights of all edges in the network. This gradient is the vector of partial derivatives of the error function w.r.t. all the weights in the network.
Basic backprop algorithm (for an arbitrary network function, could be the error function):

1. Feed in input. Calculate the primitive function values and also their derivatives. Store the derivatives in the units.

2. Run the network backwards, right to left, starting with constant 1 being submitted to the output unit. Incoming info to a node is added and the result is multiplied by the value stored in the unit. Transmit the result to the left.

3. Result collected at the input unit(s) represents the derivative of the network function w.r.t. that input.

But this just calculates the derivatives. Now, in order.to use this for learning:
Learning with backprop: Run the network forward on the given input, then use backprop to run it backwards to calculate the derivatives of the errors.
At each node, store its output for the feed-forward step, and the backprop error (the cumulative result of the backward computation up to the node) for the backprop step.
This gives us all the partial derivatives needed to calculate the gradient of the error function, w.r.t. all the weights.
Then modify the weights incrementally.

Rojas says that backprop uses only local information.
Q: So is some version of this neurally plausible?

Recurrent networks: Rojas describes how to extend the backprop approach to recurrent networks. He uses time to break cycles, as follows.
Consider a finite number T of iterations. Unwind the network, and think of the unfolding as a feed-forward network with T stages of computation.

Note: Doing this unwinding seems to depend on the units being simple function evaluators, without local state.

A complication is that weights are repeated on different copies of the same edge. In this case can simply perform backprop as usual for each edge and add the corresponding results.
Q: Is some version of this neurally plausible?

4.6 The Hopfield Model

1980s, theoretical physics approach. Uses asynchronous networks, and an energy function E.

Asynchronous network: This work assumes an asynchronous network. That doesn't mean the same thing as in distributed computing theory: instead of arbitrary timing, this uses stochastic assumptions for timing. That makes the entire system stochastic, i.e., it becomes a "stochastic dynamical system".

Equivalent formulation: Repeatedly select one unit randomly (uniformly, or possibly with different probabilities?). Let it compute its excitation and fire.

Bidirectional Associative Memory (BAM) Rojas introduces the notion of a "bidirectional associative memory (BAM)", as motivation for using an energy function.
The idea is to run the network alternatively forward and backwards, looking for a fixed point. That is, for a given pair of an x-vector and y-vector, we want to find a weight matrix that will give (x, y) as a fixed point. Hebbian learning can be used to determine an adequate matrix.

He defines a specific energy function E, see p. 341. The key property is that E assumes local minimum values at stable states of the weight matrix. He analyzes the learning algorithm using E.

Hopfield networks: Consider a clique of n units. Bidirectional connections, symmetric weights. Now the neuron states can be 1 or -1. Units have thresholds.

Can define an energy function as for BAMs. Hopfield network always finds a local min of the energy function. But some functions, like XOR, can't be computed by Hopfield networks.

Parallel combinatorics: Can use these networks to solve difficult problems in combinatorics and optimization theory. But they don't always work, e.g., for NP-complete problems. He gives some examples that give interesting results in practice.

For example, the *multiflop problem*. Looking for a binary vector of dimension n that contains exactly one 1. Sounds related to WTA, but there isn't any input here to guide the selection. A Hopfield network can work by having the unit that is first set to 1 inhibit the other units. So the symmetry is broken using time (as in Lili's work). Interestingly, they come up with the weights by systematic calculations that show how to minimize an energy function, whereas we did this work by hand. He claims that the network mirrors these calculations, systematically reducing the error function.

Q: Is this local? If so, it might be a bio-plausible way to arrive at the correct weights for a WTA network. The sort of thing we did in our papers by manual tuning. Is this approach helpful?

Other problems: Eight rooks, eight queens (doesn't always work), TSP (doesn't always work, since it's NP-complete).

4.7 Variations of the Hopfield model

Try to do better than Hopfield models in solving hard optimization problems.

Continuous Hopfield model: Model assumptions:

1. State values: Allow real values in $[0, 1]$ as possible states, not just binary.

2. Activation function is sigmoid.
   Q: Is the state actually the output of the sigmoid?

3. Timing: Dynamics still asynchronous, like Hopfield. But now changes happen with some delay.

Q: Compare with our model: Ours is synchronous. They use a particular type of continuous state (how does it compare with Lili's?). Same activation function, but no stochastic behavior here.

Rojas proves a type of convergence result.
Main difficulty is defining the energy function. He proves that the energy decreases with each state update, but that isn't enough to show convergence. Instead, he proves convergence to a small region.

This seems not to be such a big improvement, in terms of the problems the model can solve.

Introducing stochastic behavior: Another extension is to introduct some stochastic noise. The hope is that this noise can help the network to "skip out of" local minima. The best-known models here are Boltzmann machines.

Rojas discusses optimization using simulated annealing. This strategy avoids local minima of the energy function because of some (thermal) noise in the system.

A Boltzmann machine is a Hopfield network composed of n units with discrete states $x_1, x_2, ..., x_n$, each taking on values in $\{0, 1\}$.
The state of unit i is updated asynchronously according to the rule $x_i = 1$ with probability p_i, 0 with probability $1 - p_i$, where p_i is a certain sigmoid that depends on the potential and a temperature constant.
So this is discrete-state, with probability of firing derived from a sigmoid. Similar to our model, but asynchronous?

Rojas introduces a nice energy function as before. He defines the state of the system to be just the firing states of all the units. Since it is stochastic and every state can reach any other, the system doesn't absolutely guarantee convergence. So what to prove...

Ackley, Hinton, Sejnowski developed the algorithms called "Boltzmann learning". Uses gradient descent.

Boltzmann machines can be used in those cases in which basic Hopfield networks become trapped in shallow local minima of the energy function.

4.8 Kohonen networks

Now we consider unsupervised learning, where the correct output isn't known a priori. That contrasts with the previous section(s), where we are given both x and y and just want to adjust weights. So now we can't use a numerical measure of the error. However, the learning process can still yield well-defined network parameters.

Self-organization: In Chapter 5 we considered unsupervised learning in one-layer networks that identify clusters of vectors. An input vector is presented at each step, causing changes to parameters. This allows the network to build a kind of internal representation of the environment.
The best-known model that allows building such internal representations is the *topology-preserving map* model of Teuvo Kohonen.

Kohonen assumes real-valued inputs. The neural network is supposed to compute a mapping from input space A to output space B. Any input vector should lead to the firing of just one neuron in the network, yielding a "chart", or "map" of input space. Kohonen networks learn to create maps of the input space in a self-organizing way.

Rojas claims that the structures that are built in the brain are "planar". Based, presumably, on what is known about the organization of neurons in the brain. E.g., it is known that visual information is mapped as a 2-dimensional projection on the cortex, even though the information content is many-dimensional.
How is the multidimensional input projected to the more constrained two-dimensional neuronal structure?
How does the brain process these representations?

The visual cortex contains a kind of map of the visual field, some kind of deformed sphere. Neighboring regions of the visual field are processed by neighboring regions in the cortex. The surface of the visual cortex that is reserved for the processing of the information from the fovea is disproportionally large. Also, in the brain, we have proximity between the regions representing different kinds of sensory inputs, also between regions processing nearby body parts!

It seems very likely that the first representation of the world built by the brain is a topological one, in which the exterior spatial relations are mapped to similar spatial relations in the cortex.

Topological correspondence between retina and cortex is not totally genetically determined; it is known (cat experiments) that it results at least in part from sensory experience.

Kohonen's networks are arrangements of computing nodes in one-, two-, or multi-dimensional lattices. The units have lateral connections to several neighbors.

Kohonen's model: Rojas gives examples of Kohonen networks.
We can define neighboring elements in terms of the grid of computing elements in the target space (the brain). During learning, the weights of computing units and also their neighbors, get updated. This allows neighboring units to learn to react to closely related signals.
Q: Prove this.

For an example, he considers the problem of charting an n-dimensional space using a one-dimensional chain of Kohonen units. Each unit learns to specialize on different regions of input space.

A Kohonen unit is assumed to be able to compute the Euclidian distance between an input x and its weight vector w.
Learning algorithm:
Select an input vector probabilistically from the input probability space.
Present the input, and select the unit that receives the maximum excitation.
Update weight vectors using an update rule that depends on the distance in the target space.

Thus, we modify the weights for nearby units too, not just the main one that is chosen. During the learning process, gradually reduce the degree of influence on neighbors' weights.

Convergence is hard to analyze. No convergence proof exists so far, except for dimension 1.

5 Oja's work

Oja, Principal Components, Minor Components, and Linear Neural Networks Oja's earlier paper on PCA using his rule?

[[[Read the papers again, but leave it for Brabeeba?]]]