

Neural Networks for Machine Learning

Lecture 1a

Why do we need machine learning?

Geoffrey Hinton

with

Nitish Srivastava

Kevin Swersky

What is Machine Learning?

- It is very hard to write programs that solve problems like recognizing a three-dimensional object from a novel viewpoint in new lighting conditions in a cluttered scene.
 - We don't know what program to write because we don't know how it's done in our brain.
 - Even if we had a good idea about how to do it, the program might be horrendously complicated.
- It is hard to write a program to compute the probability that a credit card transaction is fraudulent.
 - There may not be any rules that are both simple and reliable. We need to combine a very large number of weak rules.
 - Fraud is a moving target. The program needs to keep changing.

The Machine Learning Approach

- Instead of writing a program by hand for each specific task, we collect lots of examples that specify the correct output for a given input.
- A machine learning algorithm then takes these examples and produces a program that does the job.
 - The program produced by the learning algorithm may look very different from a typical hand-written program. It may contain millions of numbers.
 - If we do it right, the program works for new cases as well as the ones we trained it on.
 - If the data changes the program can change too by training on the new data.
- Massive amounts of computation are now cheaper than paying someone to write a task-specific program.

Some examples of tasks best solved by learning

- Recognizing patterns:
 - Objects in real scenes
 - Facial identities or facial expressions
 - Spoken words
- Recognizing anomalies:
 - Unusual sequences of credit card transactions
 - Unusual patterns of sensor readings in a nuclear power plant
- Prediction:
 - Future stock prices or currency exchange rates
 - Which movies will a person like?

A standard example of machine learning

- A lot of genetics is done on fruit flies.
 - They are convenient because they breed fast.
 - We already know a lot about them.
- The MNIST database of hand-written digits is the the machine learning equivalent of fruit flies.
 - They are publicly available and we can learn them quite fast in a moderate-sized neural net.
 - We know a huge amount about how well various machine learning methods do on MNIST.
- We will use MNIST as our standard task.

It is very hard to say what makes a 2

0 0 0 1 1 (1 1 1, 2

2 2 2 2 2 2 2 3 3 3

3 4 4 4 4 4 5 5 5 5

6 6 7 7 7 7 7 8 8 8

8 8 8 7 9 4 9 9 9

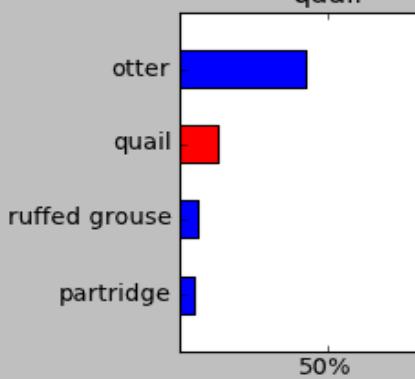
Beyond MNIST: The ImageNet task

- 1000 different object classes in 1.3 million high-resolution training images from the web.
 - Best system in 2010 competition got 47% error for its first choice and 25% error for its top 5 choices.
- Jitendra Malik (an eminent neural net sceptic) said that this competition is a good test of whether deep neural networks work well for object recognition.
 - A very deep neural net (Krizhevsky et. al. 2012) gets less than 40% error for its first choice and less than 20% for its top 5 choices (see lecture 5).

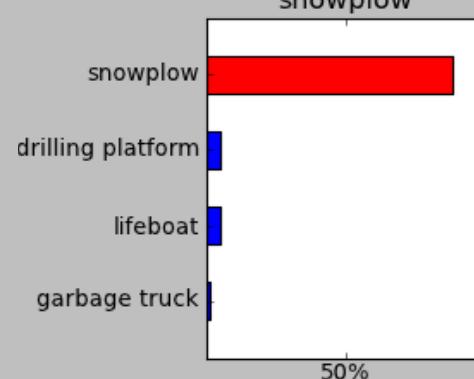
Some examples from an earlier version of the net



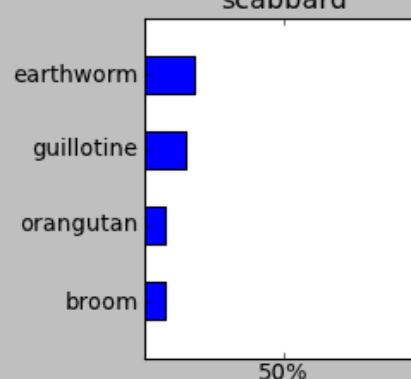
quail



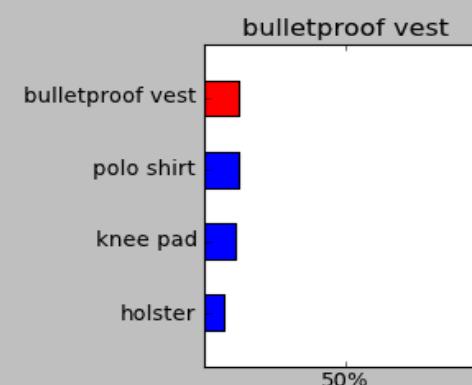
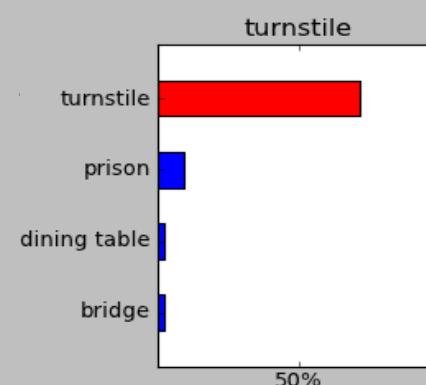
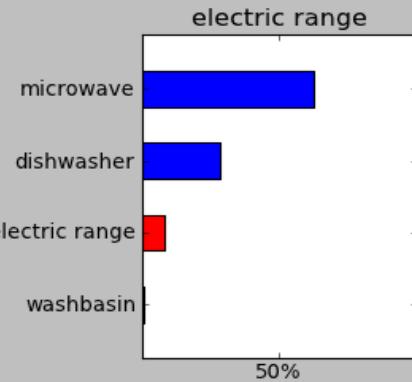
snowplow



scabbard



It can deal with a wide range of objects



It makes some really cool errors



earphone

corkscrew
lipstick
screw
ant

50%

barber chair

barber chair
drum
cello
armchair

50%

ashcan

pitcher
vacuum
measuring cup
coffeepot

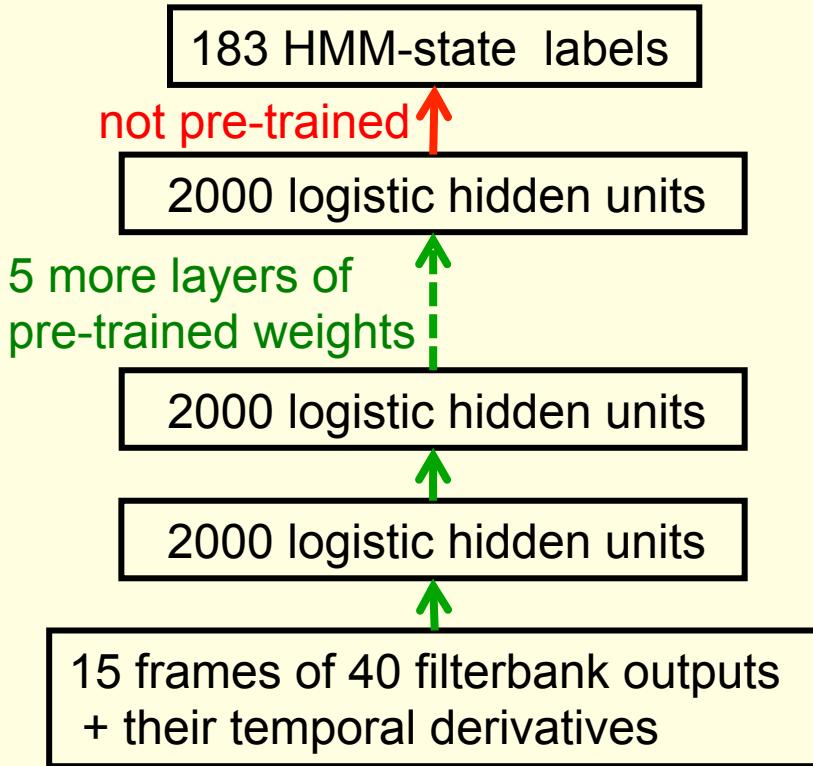
50%

The Speech Recognition Task

- A speech recognition system has several stages:
 - Pre-processing: Convert the sound wave into a vector of acoustic coefficients. Extract a new vector about every 10 milliseconds.
 - The acoustic model: Use a few adjacent vectors of acoustic coefficients to place bets on which part of which phoneme is being spoken.
 - Decoding: Find the sequence of bets that does the best job of fitting the acoustic data and also fitting a model of the kinds of things people say.
- Deep neural networks pioneered by George Dahl and Abdel-rahman Mohamed are now replacing the previous machine learning method for the acoustic model.

Phone recognition on the TIMIT benchmark

(Mohamed, Dahl, & Hinton, 2012)



- After standard post-processing using a bi-phone model, a deep net with 8 layers gets **20.7%** error rate.
- The best previous speaker-independent result on TIMIT was **24.4%** and this required averaging several models.
- Li Deng (at MSR) realised that this result could change the way speech recognition was done.

Word error rates from MSR, IBM, & Google

(Hinton et. al. IEEE Signal Processing Magazine, Nov 2012)

The task	Hours of training data	Deep neural network	Gaussian Mixture Model	GMM with more data
Switchboard (Microsoft Research)	309	18.5%	27.4%	18.6% (2000 hrs)
English broadcast news (IBM)	50	17.5%	18.8%	
Google voice search (android 4.1)	5,870	12.3% (and falling)		16.0% (>>5,870 hrs)

Neural Networks for Machine Learning

Lecture 1b What are neural networks?

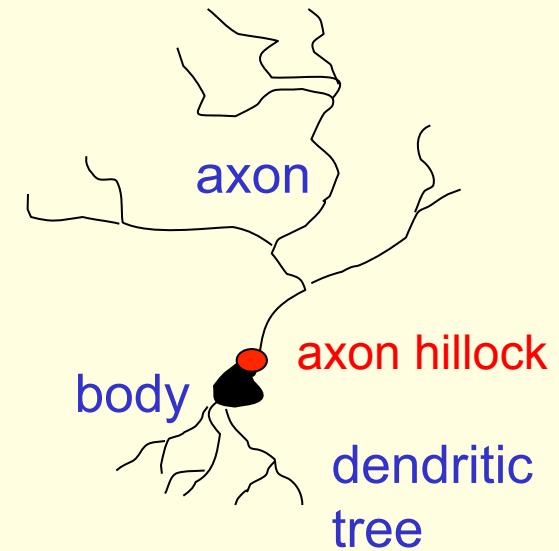
Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

Reasons to study neural computation

- To understand how the brain actually works.
 - Its very big and very complicated and made of stuff that dies when you poke it around. So we need to use computer simulations.
- To understand a style of parallel computation inspired by neurons and their adaptive connections.
 - Very different style from sequential computation.
 - should be good for things that brains are good at (e.g. vision)
 - Should be bad for things that brains are bad at (e.g. 23×71)
- To solve practical problems by using novel learning algorithms inspired by the brain (this course)
 - Learning algorithms can be very useful even if they are not how the brain actually works.

A typical cortical neuron

- Gross physical structure:
 - There is one axon that branches
 - There is a dendritic tree that collects input from other neurons.
- Axons typically contact dendritic trees at synapses
 - A spike of activity in the axon causes charge to be injected into the post-synaptic neuron.
- Spike generation:
 - There is an **axon hillock** that generates outgoing spikes whenever enough charge has flowed in at synapses to depolarize the cell membrane.



Synapses

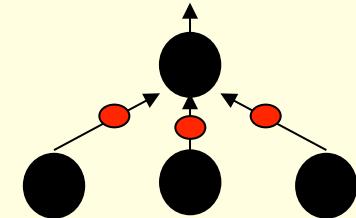
- When a spike of activity travels along an axon and arrives at a synapse it causes vesicles of transmitter chemical to be released.
 - There are several kinds of transmitter.
- The transmitter molecules diffuse across the synaptic cleft and bind to receptor molecules in the membrane of the post-synaptic neuron thus changing their shape.
 - This opens up holes that allow specific ions in or out.

How synapses adapt

- The effectiveness of the synapse can be changed:
 - vary the number of vesicles of transmitter.
 - vary the number of receptor molecules.
- Synapses are slow, but they have advantages over RAM
 - They are very small and very low-power.
 - They adapt using locally available signals
 - But what rules do they use to decide how to change?

How the brain works on one slide!

- Each neuron receives inputs from other neurons
 - A few neurons also connect to receptors.
 - Cortical neurons use spikes to communicate.
- The effect of each input line on the neuron is controlled by a synaptic weight
 - The weights can be positive or negative.
- The synaptic weights adapt so that the whole network learns to perform useful computations
 - Recognizing objects, understanding language, making plans, controlling the body.
- You have about 10^{11} neurons each with about 10^4 weights.
 - A huge number of weights can affect the computation in a very short time. Much better bandwidth than a workstation.



Modularity and the brain

- Different bits of the cortex do different things.
 - Local damage to the brain has specific effects.
 - Specific tasks increase the blood flow to specific regions.
- But cortex looks pretty much the same all over.
 - Early brain damage makes functions relocate.
- Cortex is made of general purpose stuff that has the ability to turn into special purpose hardware in response to experience.
 - This gives rapid parallel computation plus flexibility.
 - Conventional computers get flexibility by having stored sequential programs, but this requires very fast central processors to perform long sequential computations.

Neural Networks for Machine Learning

Lecture 1c Some simple models of neurons

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

Idealized neurons

- To model things we have to idealize them (e.g. atoms)
 - Idealization removes complicated details that are not essential for understanding the main principles.
 - It allows us to apply mathematics and to make analogies to other, familiar systems.
 - Once we understand the basic principles, it's easy to add complexity to make the model more faithful.
- It is often worth understanding models that are known to be wrong (but we must not forget that they are wrong!)
 - E.g. neurons that communicate real values rather than discrete spikes of activity.

Linear neurons

- These are simple but computationally limited
 - If we can make them learn we **may** get insight into more complicated neurons.

$$y = b + \sum_i x_i w_i$$

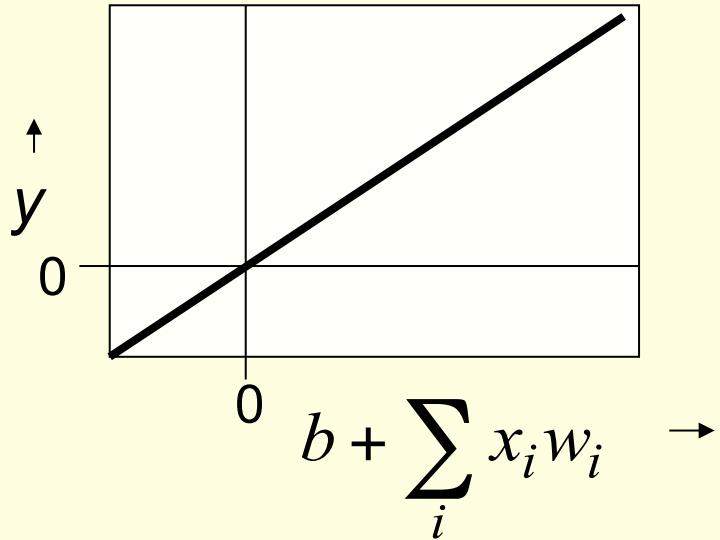
Annotations in red:

- bias: arrow pointing to the constant term b .
- i^{th} input: arrow pointing to the i^{th} term $x_i w_i$.
- output: arrow pointing to the result y .
- index over input connections: arrow pointing to the index i under the summation symbol.
- weight on i^{th} input: arrow pointing to the product $x_i w_i$.

Linear neurons

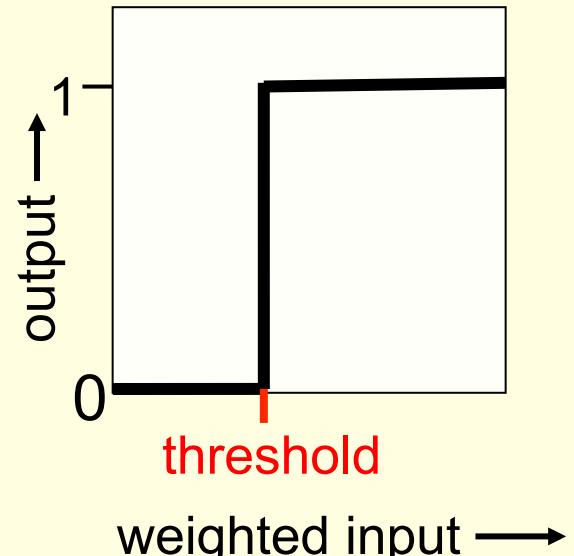
- These are simple but computationally limited
 - If we can make them learn we **may** get insight into more complicated neurons.

$$y = b + \sum_i x_i w_i$$



Binary threshold neurons

- McCulloch-Pitts (1943): influenced Von Neumann.
 - First compute a weighted sum of the inputs.
 - Then send out a fixed size spike of activity if the weighted sum exceeds a threshold.
 - McCulloch and Pitts thought that each spike is like the truth value of a proposition and each neuron combines truth values to compute the truth value of another proposition!



Binary threshold neurons

- There are two equivalent ways to write the equations for a binary threshold neuron:

$$z = \sum_i x_i w_i$$

$$y = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

$$\theta = -b$$

$$z = b + \sum_i x_i w_i$$

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Rectified Linear Neurons

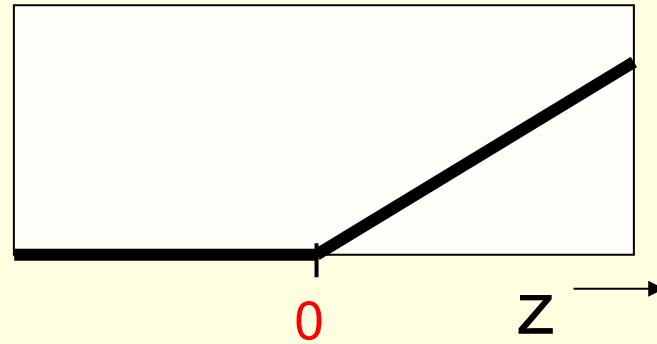
(sometimes called linear threshold neurons)

They compute a **linear** weighted sum of their inputs.
The output is a **non-linear** function of the total input.

$$z = b + \sum_i x_i w_i$$

$$y = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

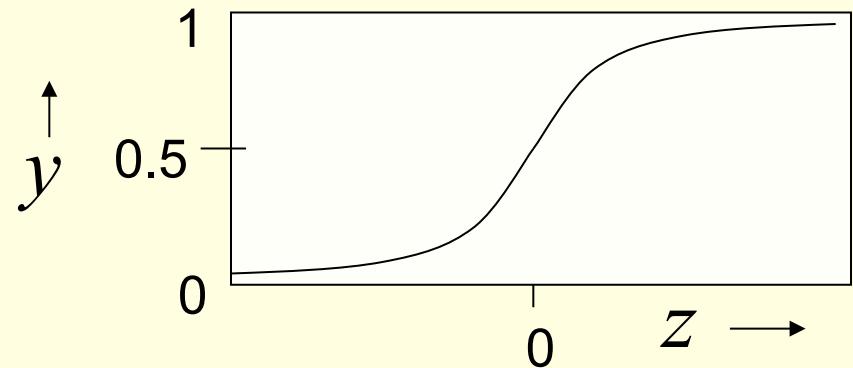
↑
y



Sigmoid neurons

- These give a real-valued output that is a smooth and bounded function of their total input.
 - Typically they use the logistic function
 - They have nice derivatives which make learning easy (see lecture 3).

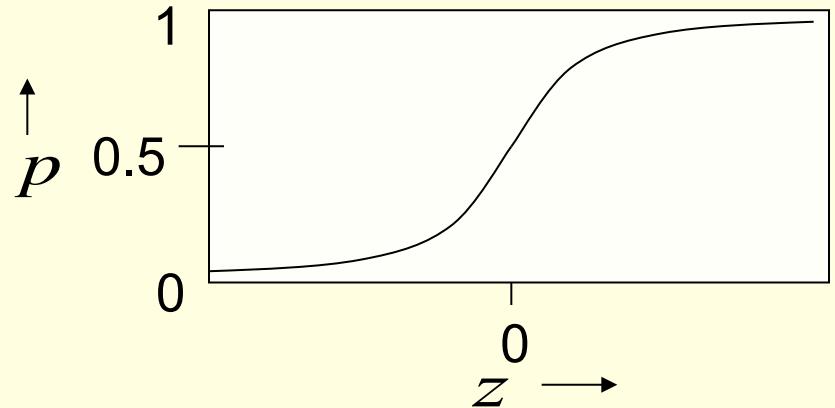
$$z = b + \sum_i x_i w_i \quad y = \frac{1}{1 + e^{-z}}$$



Stochastic binary neurons

- These use the same equations as logistic units.
 - But they treat the output of the logistic as the probability of producing a spike in a short time window.
- We can do a similar trick for rectified linear units:
 - The output is treated as the Poisson rate for spikes.

$$z = b + \sum_i x_i w_i \quad p(s=1) = \frac{1}{1 + e^{-z}}$$



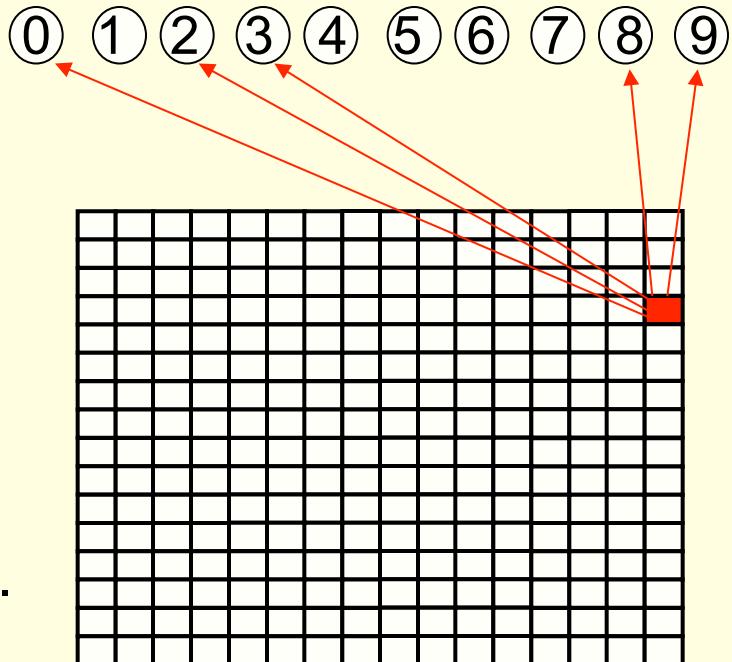
Neural Networks for Machine Learning

Lecture 1d A simple example of learning

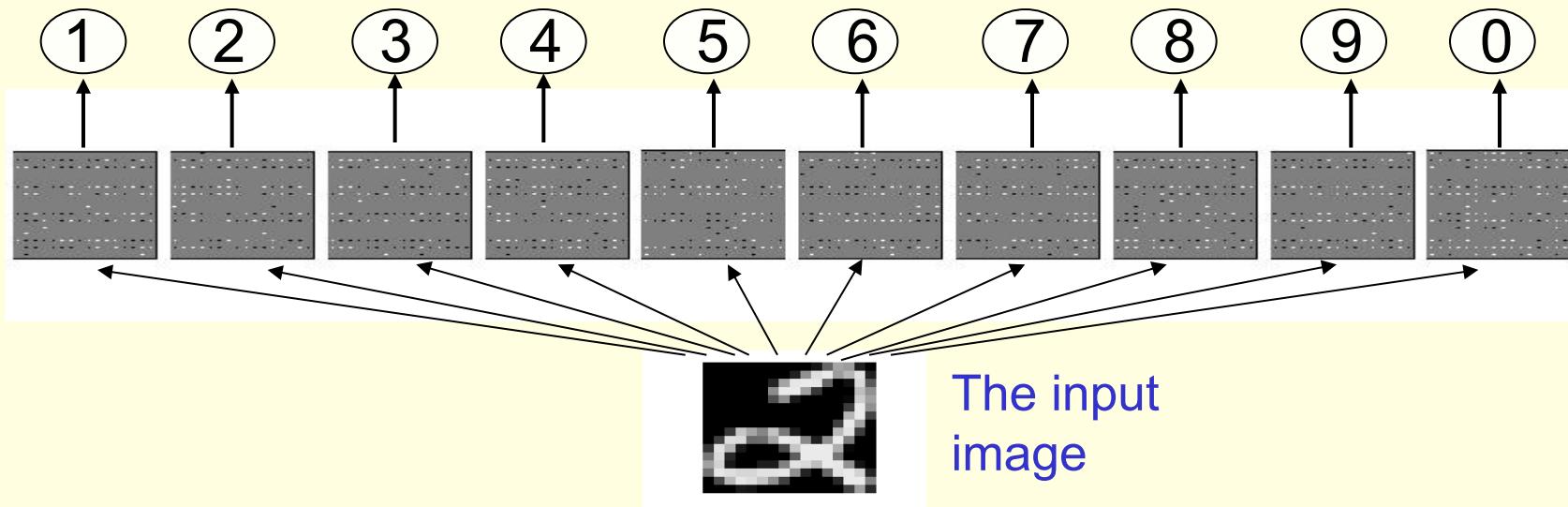
Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

A very simple way to recognize handwritten shapes

- Consider a neural network with two layers of neurons.
 - neurons in the top layer represent known shapes.
 - neurons in the bottom layer represent pixel intensities.
- A pixel gets to vote if it has ink on it.
 - Each inked pixel can vote for several different shapes.
- The shape that gets the most votes wins.



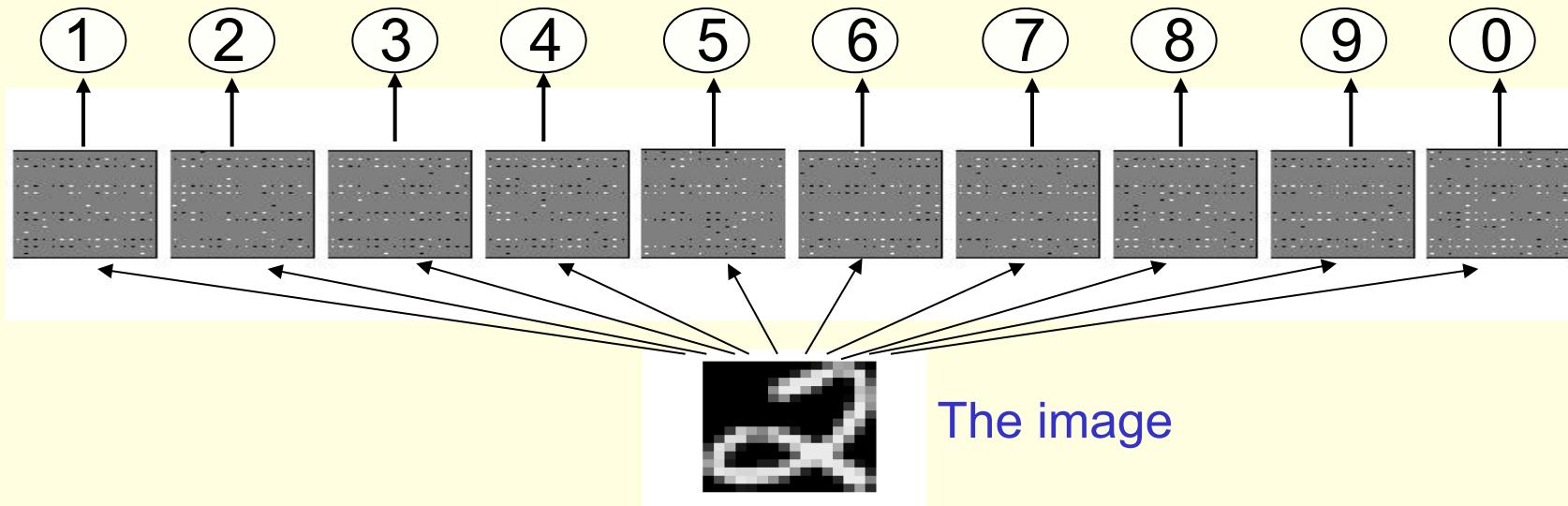
How to display the weights



Give each output unit its own “map” of the input image and display the weight coming from each pixel in the location of that pixel in the map.

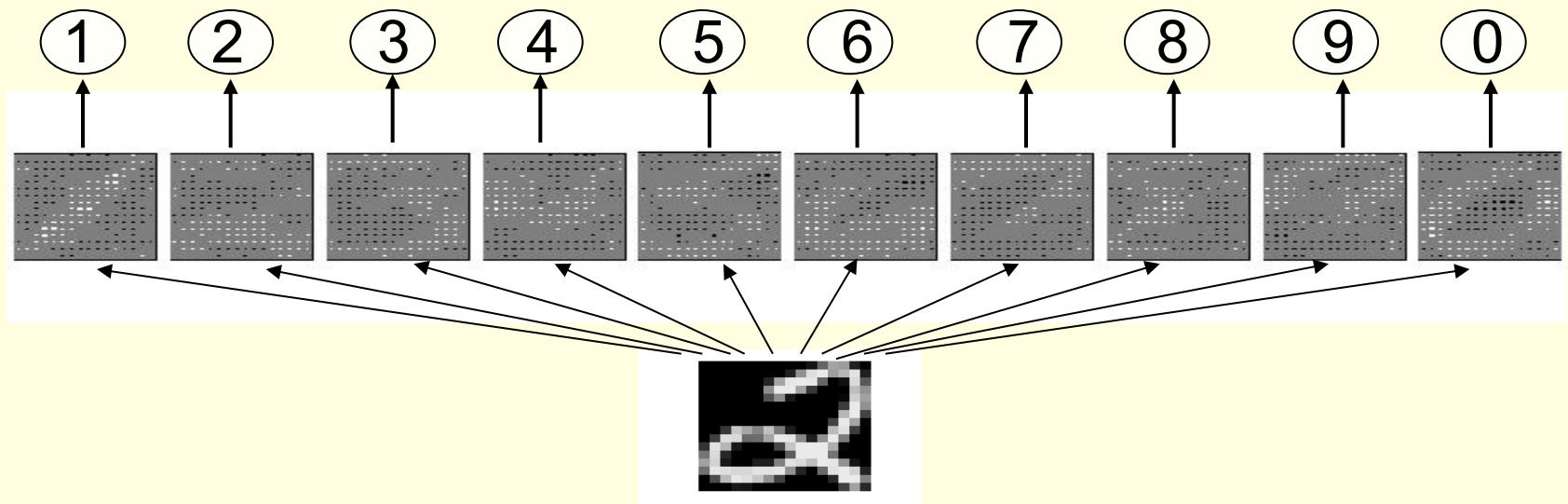
Use a black or white blob with the area representing the magnitude of the weight and the color representing the sign.

How to learn the weights

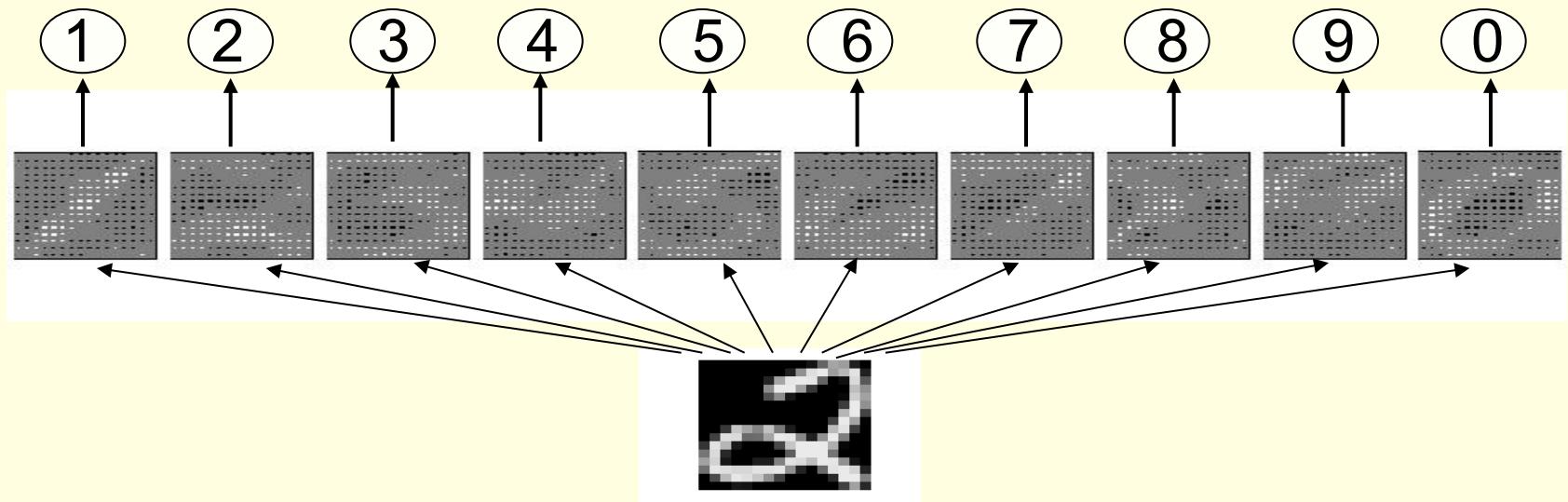


Show the network an image and **increment** the weights from active pixels to the correct class.

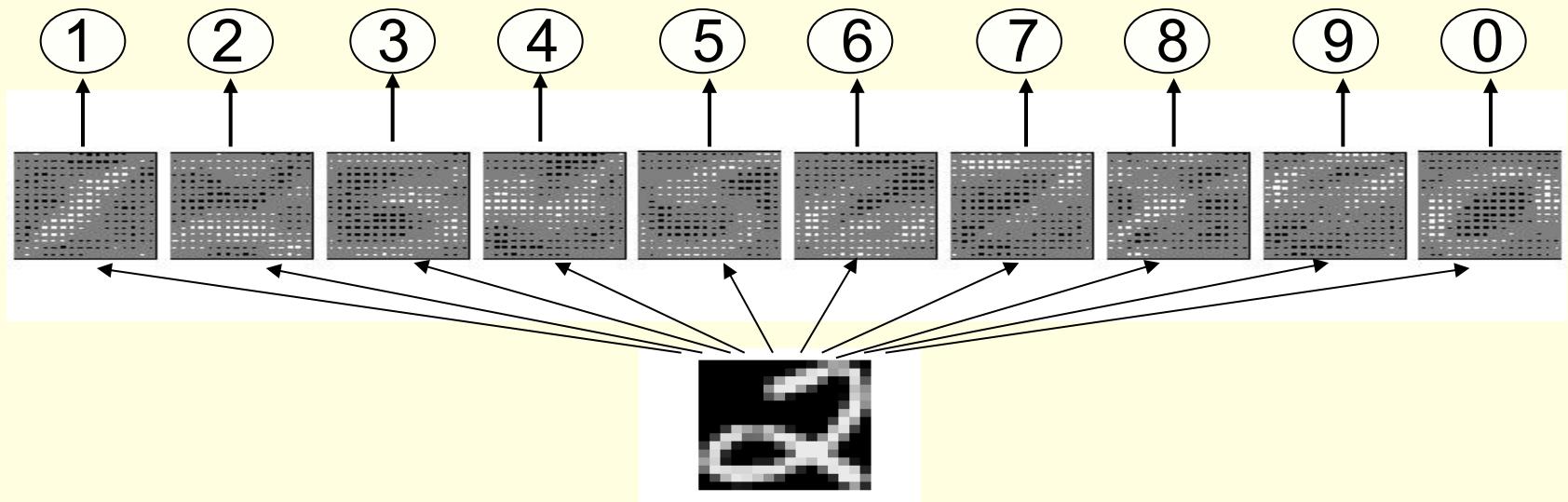
Then **decrement** the weights from active pixels to whatever class the network guesses.



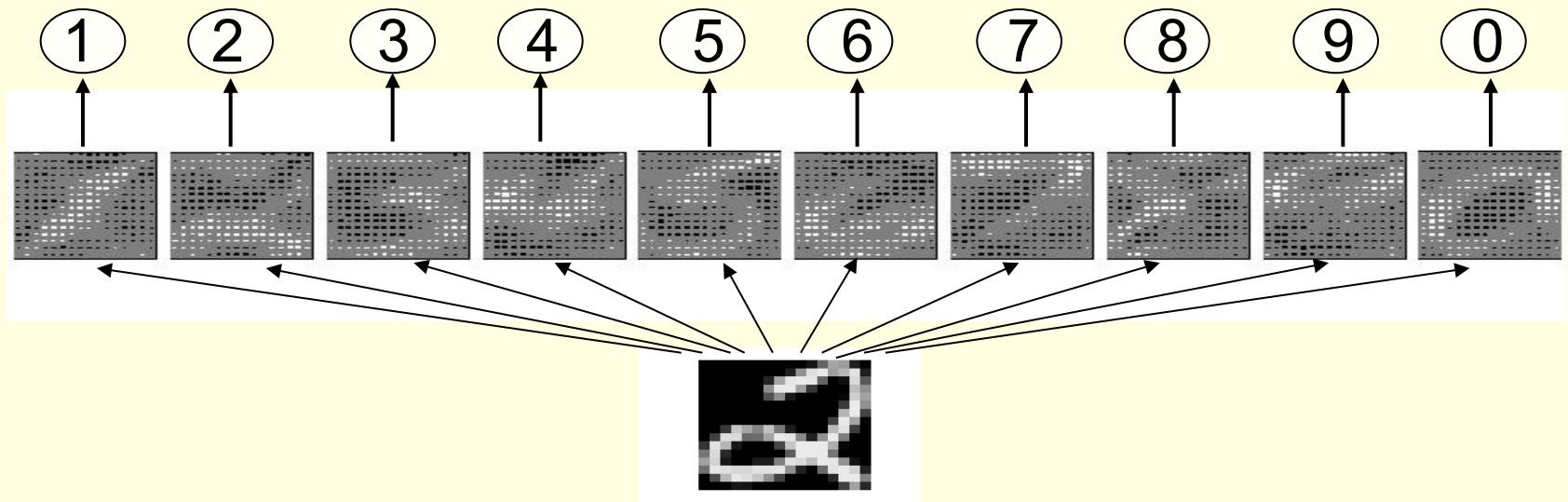
The image



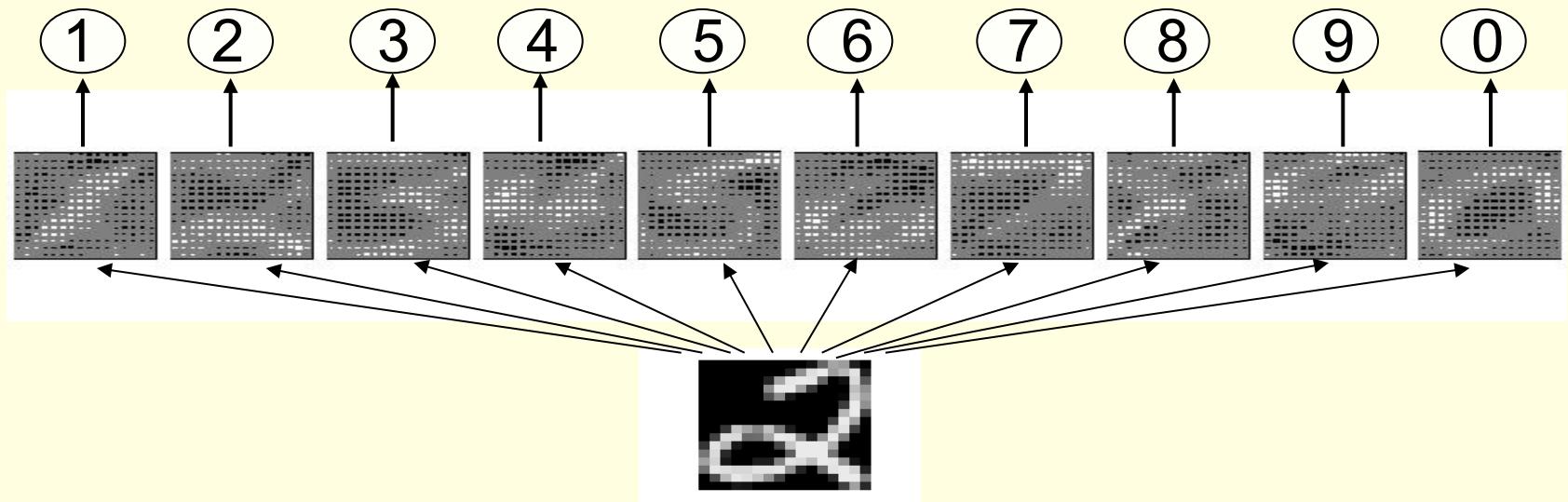
The image



The image

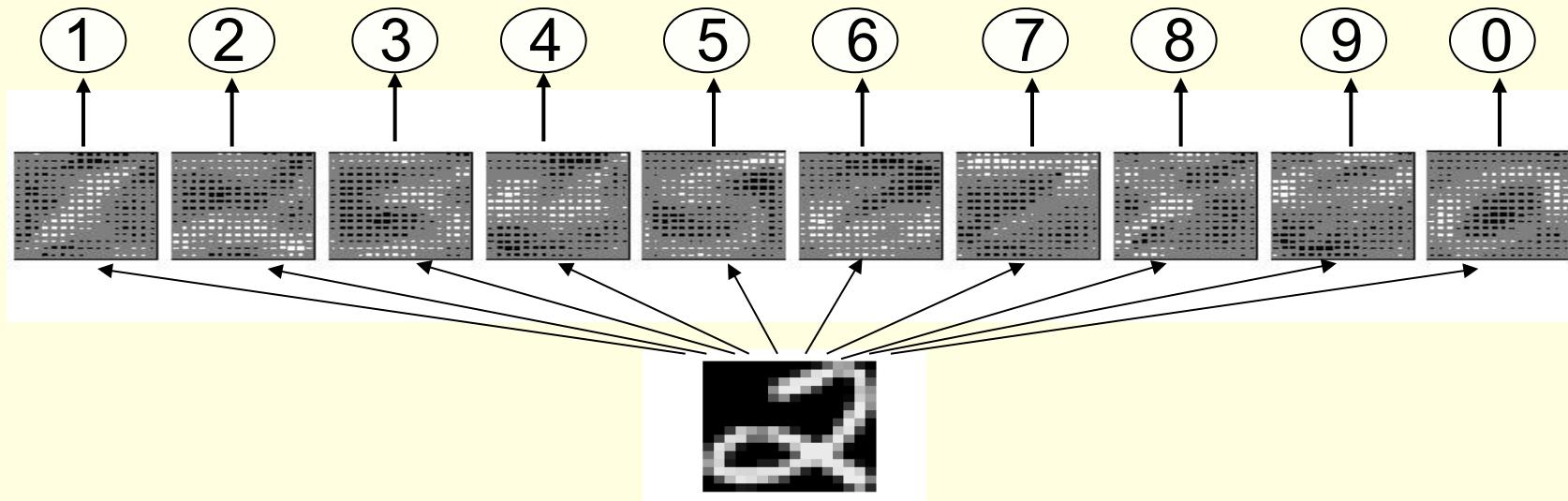


The image



The image

The learned weights



The details of the learning algorithm will be explained in future lectures.

Why the simple learning algorithm is insufficient

- A two layer network with a single winner in the top layer is equivalent to having a rigid template for each shape.
 - The winner is the template that has the biggest overlap with the ink.
- The ways in which hand-written digits vary are much too complicated to be captured by simple template matches of whole shapes.
 - To capture all the allowable variations of a digit we need to learn the features that it is composed of.

Examples of handwritten digits that can be recognized correctly the first time they are seen

0 0 0 1 1 (1 1 1, 2

2 2 2 2 2 2 3 3 3

3 4 4 4 4 4 5 5 5

6 6 7 7 7 7 8 8 8

8 8 8 7 9 4 9 9 9

Neural Networks for Machine Learning

Lecture 1e Three types of learning

Geoffrey Hinton

with

Nitish Srivastava

Kevin Swersky

Types of learning task

- Supervised learning
 - Learn to predict an output when given an input vector.
- Reinforcement learning
 - Learn to select an action to maximize payoff.
- Unsupervised learning
 - Discover a good internal representation of the input.

Two types of supervised learning

- Each training case consists of an input vector x and a target output t .
- Regression: The target output is a real number or a whole vector of real numbers.
 - The price of a stock in 6 months time.
 - The temperature at noon tomorrow.
- Classification: The target output is a class label.
 - The simplest case is a choice between 1 and 0.
 - We can also have multiple alternative labels.

How supervised learning typically works

- We start by choosing a model-class: $y = f(\mathbf{x}; \mathbf{W})$
 - A model-class, f , is a way of using some numerical parameters, \mathbf{W} , to map each input vector, \mathbf{x} , into a predicted output y .
- Learning usually means adjusting the parameters to reduce the discrepancy between the target output, t , on each training case and the actual output, y , produced by the model.
 - For regression, $\frac{1}{2}(y - t)^2$ is often a sensible measure of the discrepancy.
 - For classification there are other measures that are generally more sensible (they also work better).

Reinforcement learning

- In reinforcement learning, the output is an action or sequence of actions and the only supervisory signal is an occasional scalar reward.
 - The goal in selecting each action is to maximize the expected sum of the future rewards.
 - We usually use a discount factor for delayed rewards so that we don't have to look too far into the future.
- Reinforcement learning is difficult:
 - The rewards are typically delayed so its hard to know where we went wrong (or right).
 - A scalar reward does not supply much information.
- This course cannot cover everything and reinforcement learning is one of the important topics we will not cover.

Unsupervised learning

- For about 40 years, unsupervised learning was largely ignored by the machine learning community
 - Some widely used definitions of machine learning actually excluded it.
 - Many researchers thought that clustering was the only form of unsupervised learning.
- It is hard to say what the aim of unsupervised learning is.
 - One major aim is to create an internal representation of the input that is useful for subsequent supervised or reinforcement learning.
 - You can compute the distance to a surface by using the disparity between two images. But you don't want to learn to compute disparities by stubbing your toe thousands of times.

Other goals for unsupervised learning

- It provides a compact, low-dimensional representation of the input.
 - High-dimensional inputs typically live on or near a low-dimensional manifold (or several such manifolds).
 - Principal Component Analysis is a widely used linear method for finding a low-dimensional representation.
- It provides an economical high-dimensional representation of the input in terms of learned features.
 - Binary features are economical.
 - So are real-valued features that are nearly all zero.
- It finds sensible clusters in the input.
 - This is an example of a **very** sparse code in which only one of the features is non-zero.

Neural Networks for Machine Learning

Lecture 2a

An overview of the main types of neural network architecture

Geoffrey Hinton

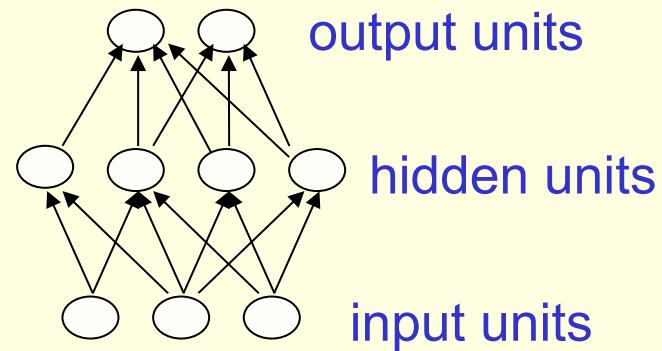
with

Nitish Srivastava

Kevin Swersky

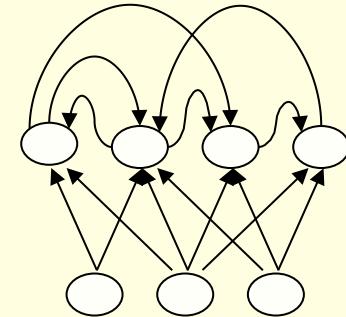
Feed-forward neural networks

- These are the commonest type of neural network in practical applications.
 - The first layer is the input and the last layer is the output.
 - If there is more than one hidden layer, we call them “deep” neural networks.
- They compute a series of transformations that change the similarities between cases.
 - The activities of the neurons in each layer are a non-linear function of the activities in the layer below.



Recurrent networks

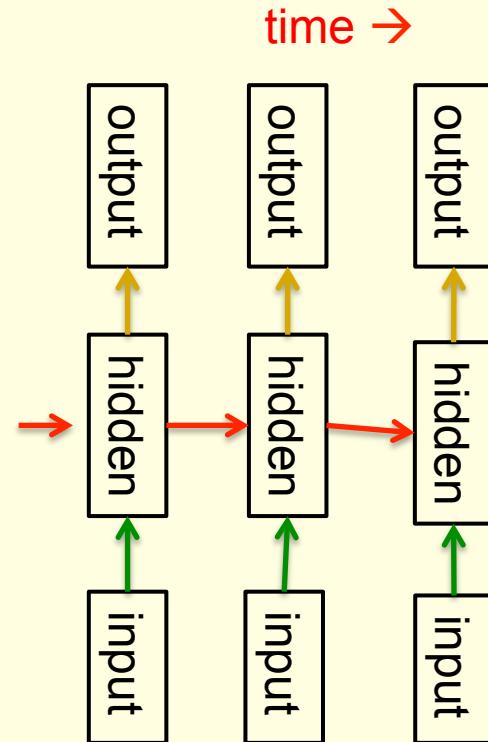
- These have directed cycles in their connection graph.
 - That means you can sometimes get back to where you started by following the arrows.
- They can have complicated dynamics and this can make them very difficult to train.
 - There is a lot of interest at present in finding efficient ways of training recurrent nets.
- They are more biologically realistic.



Recurrent nets with multiple hidden layers are just a special case that has some of the $\text{hidden} \rightarrow \text{hidden}$ connections missing.

Recurrent neural networks for modeling sequences

- Recurrent neural networks are a very natural way to model sequential data:
 - They are equivalent to very deep nets with one hidden layer per time slice.
 - Except that they use the same weights at every time slice and they get input at every time slice.
- They have the ability to remember information in their hidden state for a long time.
 - But its very hard to train them to use this potential.



An example of what recurrent neural nets can now do (to whet your interest!)

- Ilya Sutskever (2011) trained a special type of recurrent neural net to predict the next character in a sequence.
- After training for a long time on a string of half a billion characters from English Wikipedia, he got it to generate new text.
 - It generates by predicting the probability distribution for the next character and then sampling a character from this distribution.
 - The next slide shows an example of the kind of text it generates. Notice how much it knows!

Some text generated **one character at a time** by Ilya Sutskever's recurrent neural network

In 1974 Northern Denver had been overshadowed by CNL, and several Irish intelligence agencies in the Mediterranean region. However, on the Victoria, Kings Hebrew stated that Charles decided to escape during an alliance. The mansion house was completed in 1882, the second in its bridge are omitted, while closing is the proton reticulum composed below it aims, such that it is the blurring of appearing on any well-paid type of box printer.

Symmetrically connected networks

- These are like recurrent networks, but the connections between units are symmetrical (they have the same weight in both directions).
 - John Hopfield (and others) realized that symmetric networks are much easier to analyze than recurrent networks.
 - They are also more restricted in what they can do. because they obey an energy function.
 - For example, they cannot model cycles.
- Symmetrically connected nets without hidden units are called “Hopfield nets”.

Symmetrically connected networks with hidden units

- These are called “Boltzmann machines”.
 - They are much more powerful models than Hopfield nets.
 - They are less powerful than recurrent neural networks.
 - They have a beautifully simple learning algorithm.
- We will cover Boltzmann machines towards the end of the course.

Neural Networks for Machine Learning

Lecture 2b Perceptrons:

The first generation of neural networks

Geoffrey Hinton

with

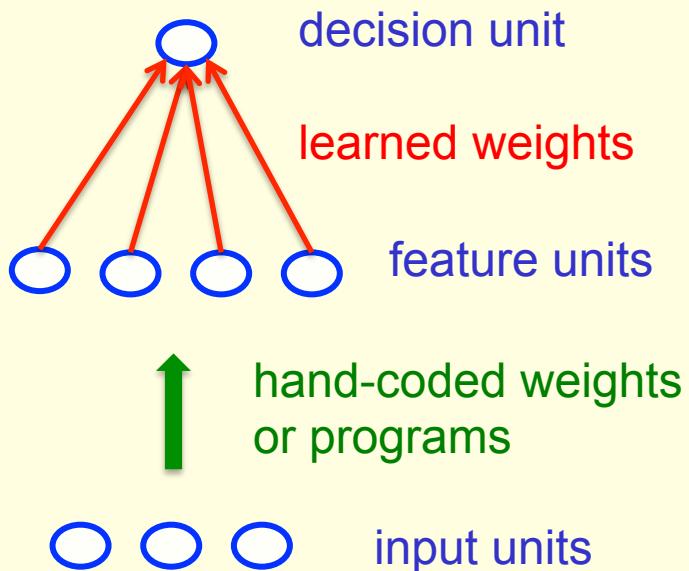
Nitish Srivastava

Kevin Swersky

The standard paradigm for statistical pattern recognition

1. Convert the raw input vector into a vector of feature activations.
Use hand-written programs based on common-sense to define the features.
2. Learn how to weight each of the feature activations to get a single scalar quantity.
3. If this quantity is above some threshold, decide that the input vector is a positive example of the target class.

The standard Perceptron architecture



The history of perceptrons

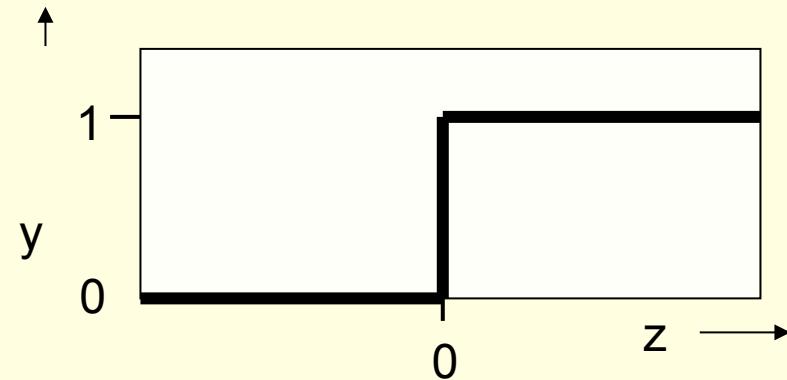
- They were popularised by Frank Rosenblatt in the early 1960's.
 - They appeared to have a very powerful learning algorithm.
 - Lots of grand claims were made for what they could learn to do.
- In 1969, Minsky and Papert published a book called “Perceptrons” that analysed what they could do and showed their limitations.
 - Many people thought these limitations applied to all neural network models.
- The perceptron learning procedure is still widely used today for tasks with enormous feature vectors that contain many millions of features.

Binary threshold neurons (decision units)

- McCulloch-Pitts (1943)
 - First compute a weighted sum of the inputs from other neurons (plus a bias).
 - Then output a 1 if the weighted sum exceeds zero.

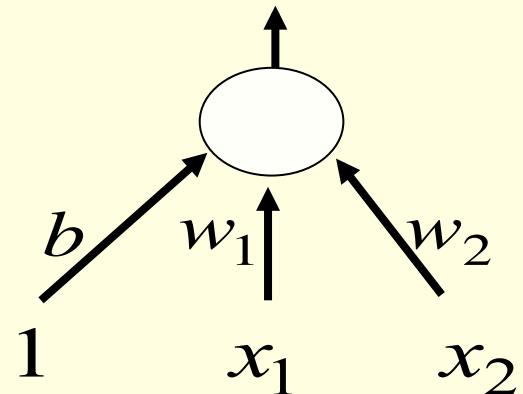
$$z = b + \sum_i x_i w_i$$

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$



How to learn biases using the same rule as we use for learning weights

- A threshold is equivalent to having a negative bias.
- We can avoid having to figure out a separate learning rule for the bias by using a trick:
 - A bias is exactly equivalent to a weight on an extra input line that always has an activity of 1.
 - We can now learn a bias as if it were a weight.



The perceptron convergence procedure: Training binary output neurons as classifiers

- Add an extra component with value 1 to each input vector. The “bias” weight on this component is minus the threshold. Now we can forget the threshold.
- Pick training cases using any policy that ensures that every training case will keep getting picked.
 - If the output unit is correct, leave its weights alone.
 - If the output unit incorrectly outputs a zero, add the input vector to the weight vector.
 - If the output unit incorrectly outputs a 1, subtract the input vector from the weight vector.
- This is guaranteed to find a set of weights that gets the right answer for all the training cases **if any such set exists.**

Neural Networks for Machine Learning

Lecture 2c A geometrical view of perceptrons

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

Warning!

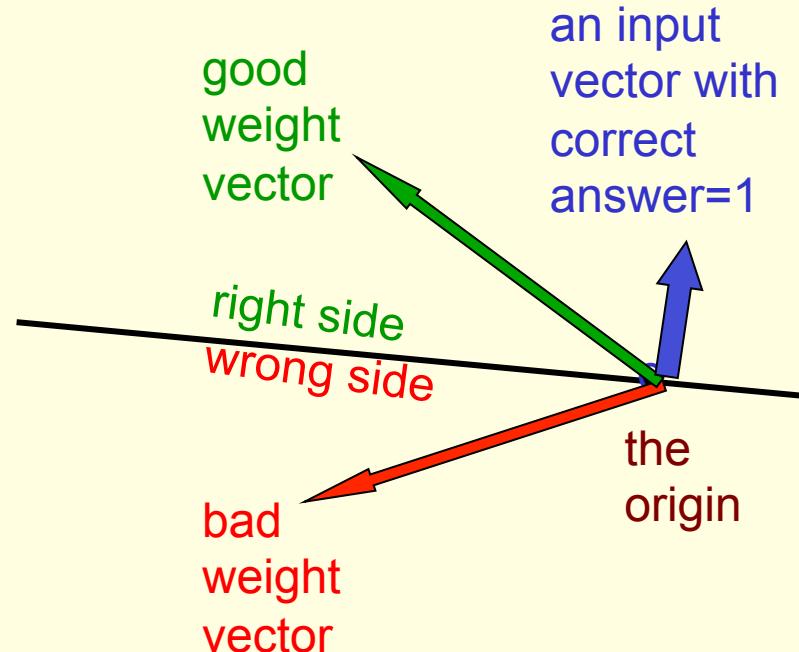
- For non-mathematicians, this is going to be tougher than the previous material.
 - You may have to spend a long time studying the next two parts.
- If you are not used to thinking about hyper-planes in high-dimensional spaces, now is the time to learn.
- To deal with hyper-planes in a 14-dimensional space, visualize a 3-D space and say “fourteen” to yourself very loudly. Everyone does it.
 - But remember that going from 13-D to 14-D creates as much extra complexity as going from 2-D to 3-D.

Weight-space

- This space has one dimension per weight.
- A point in the space represents a particular setting of all the weights.
- Assuming that we have eliminated the threshold, each training case can be represented as a hyperplane through the origin.
 - The weights must lie on one side of this hyper-plane to get the answer correct.

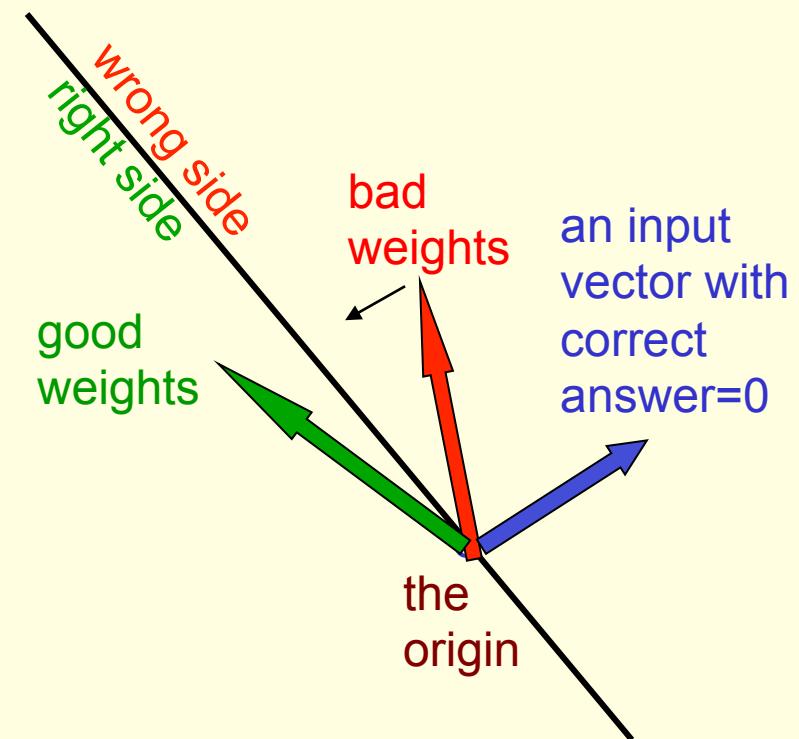
Weight space

- Each training case defines a plane (shown as a black line)
 - The plane goes through the origin and is perpendicular to the input vector.
 - On one side of the plane the output is **wrong** because the scalar product of the weight vector with the input vector has the wrong sign.



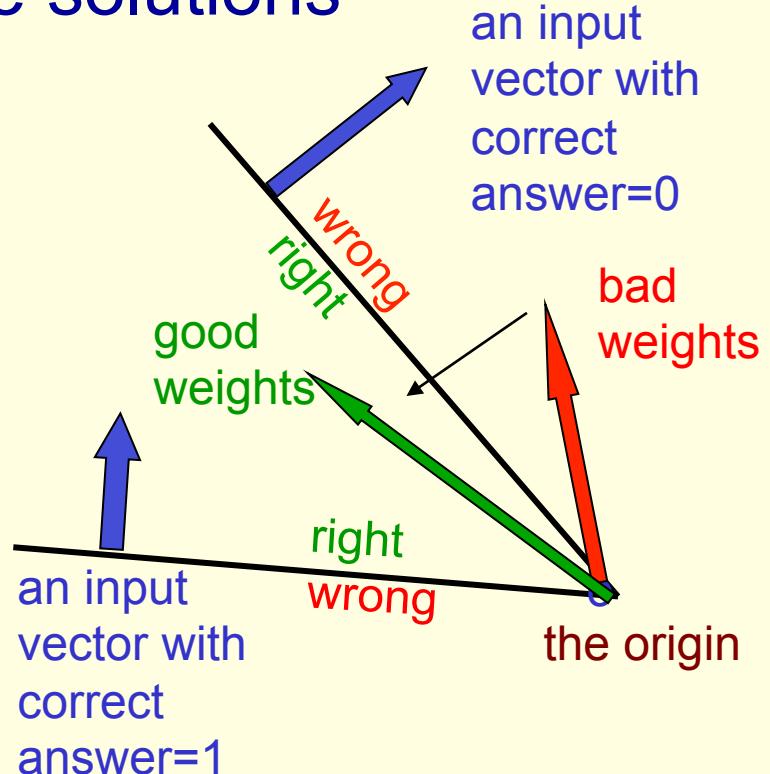
Weight space

- Each training case defines a plane (shown as a black line)
 - The plane goes through the origin and is perpendicular to the input vector.
 - On one side of the plane the output is **wrong** because the scalar product of the weight vector with the input vector has the wrong sign.



The cone of feasible solutions

- To get all training cases right we need to find a point on the right side of all the planes.
 - There may not be any such point!
- If there are any weight vectors that get the right answer for all cases, they lie in a hyper-cone with its apex at the origin.
 - So the average of two good weight vectors is a good weight vector.
 - The problem is convex.



Neural Networks for Machine Learning

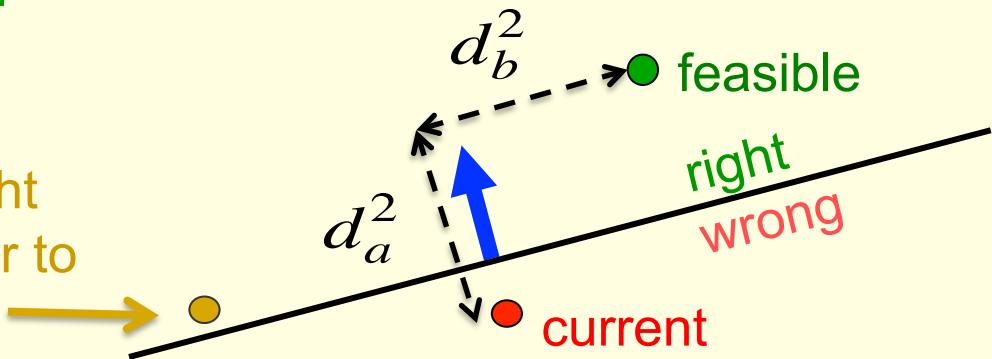
Lecture 2d Why the learning works

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

Why the learning procedure works (first attempt)

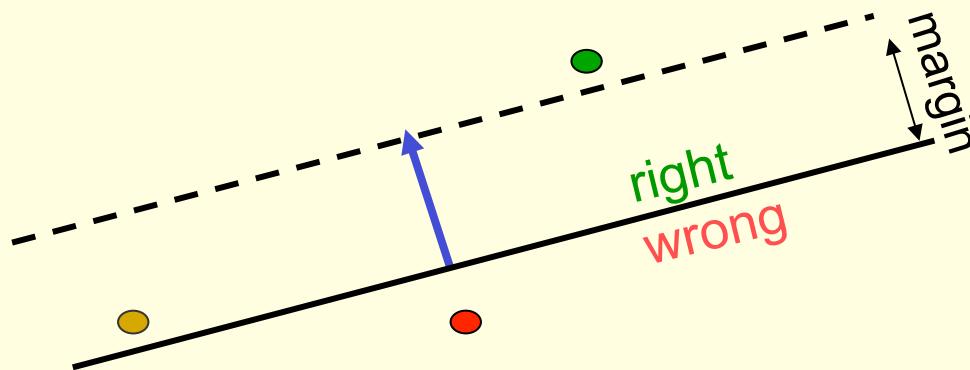
- Consider the squared distance $d_a^2 + d_b^2$ between any feasible weight vector and the current weight vector.
 - Hopeful claim: Every time the perceptron makes a mistake, the learning algorithm moves the current weight vector closer to all feasible weight vectors.

Problem case: The weight vector may not get closer to this feasible vector!



Why the learning procedure works

- So consider “generously feasible” weight vectors that lie within the feasible region by a margin at least as great as the length of the input vector that defines each constraint plane.
 - Every time the perceptron makes a mistake, the squared distance to all of these generously feasible weight vectors is always decreased by at least the squared length of the update vector.



Informal sketch of proof of convergence

- Each time the perceptron makes a mistake, the current weight vector moves to decrease its squared distance from every weight vector in the “generously feasible” region.
- The squared distance decreases by at least the squared length of the input vector.
- So after a finite number of mistakes, the weight vector must lie in the feasible region **if this region exists.**

Neural Networks for Machine Learning

Lecture 2e What perceptrons can't do

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

The limitations of Perceptrons

- If you are allowed to choose the features by hand and if you use enough features, you can do almost anything.
 - For binary input vectors, we can have a separate feature unit for each of the exponentially many binary vectors and so we can make any possible discrimination on binary input vectors.
 - This type of table look-up won't generalize.
- But once the hand-coded features have been determined, there are very strong limitations on what a perceptron can learn.

What binary threshold neurons cannot do

- A binary threshold output unit cannot even tell if two single bit features are the same!

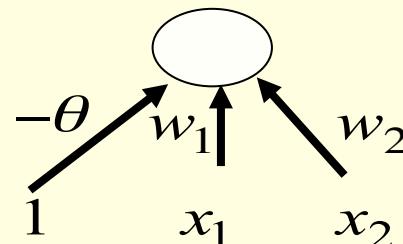
Positive cases (same): $(1,1) \rightarrow 1; (0,0) \rightarrow 1$

Negative cases (different): $(1,0) \rightarrow 0; (0,1) \rightarrow 0$

- The four input-output pairs give four inequalities that are impossible to satisfy:

$$w_1 + w_2 \geq \theta, \quad 0 \geq \theta$$

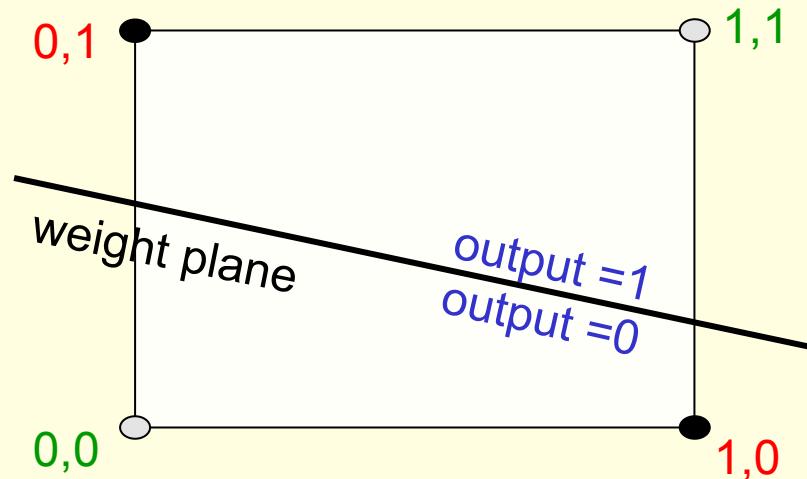
$$w_1 < \theta, \quad w_2 < \theta$$



A geometric view of what binary threshold neurons cannot do

Imagine “data-space” in which the axes correspond to components of an input vector.

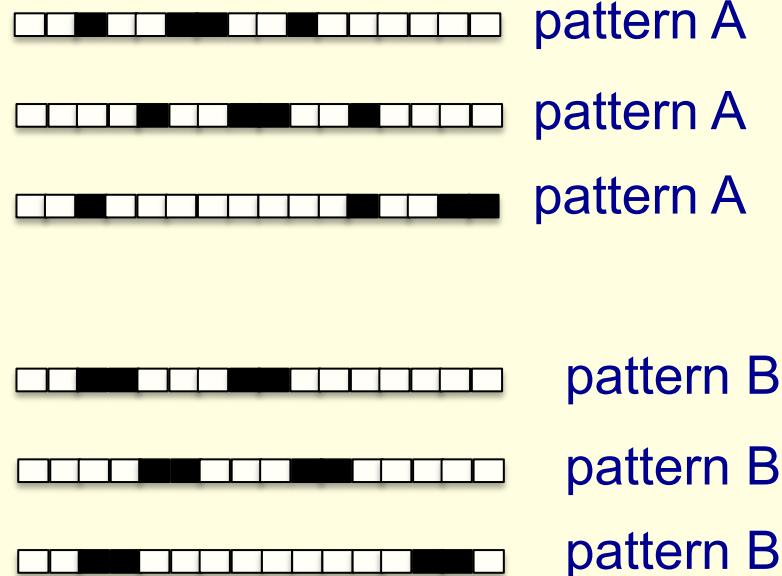
- Each input vector is a point in this space.
- A weight vector defines a plane in data-space.
- The weight plane is perpendicular to the weight vector and misses the origin by a distance equal to the threshold.



The positive and negative cases cannot be separated by a plane

Discriminating simple patterns under translation with wrap-around

- Suppose we just use pixels as the features.
- Can a binary threshold unit discriminate between different patterns that have the same number of on pixels?
 - Not if the patterns can translate with wrap-around!



Sketch of a proof that a binary decision unit cannot discriminate patterns with the same number of on pixels (assuming translation with wraparound)

- For pattern A, use training cases in all possible translations.
 - Each pixel will be activated by 4 different translations of pattern A.
 - So the total input received by the decision unit over all these patterns will be four times the sum of all the weights.
- For pattern B, use training cases in all possible translations.
 - Each pixel will be activated by 4 different translations of pattern B.
 - So the total input received by the decision unit over all these patterns will be four times the sum of all the weights.
- But to discriminate correctly, every single case of pattern A must provide more input to the decision unit than every single case of pattern B.
 - This is impossible if the sums over cases are the same.

Why this result is devastating for Perceptrons

- The whole point of pattern recognition is to recognize patterns despite transformations like translation.
- Minsky and Papert's "Group Invariance Theorem" says that the part of a Perceptron that learns cannot learn to do this if the transformations form a group.
 - Translations with wrap-around form a group.
- To deal with such transformations, a Perceptron needs to use multiple feature units to recognize transformations of informative sub-patterns.
 - So the tricky part of pattern recognition must be solved by the hand-coded feature detectors, not the learning procedure.

Learning with hidden units

- Networks without hidden units are very limited in the input-output mappings they can learn to model.
 - More layers of linear units do not help. Its still linear.
 - Fixed output non-linearities are not enough.
- We need multiple layers of **adaptive**, non-linear hidden units. But how can we train such nets?
 - We need an efficient way of adapting **all** the weights, not just the last layer. This is hard.
 - Learning the weights going into hidden units is equivalent to learning features.
 - This is difficult because nobody is telling us directly what the hidden units should do.

Neural Networks for Machine Learning

Lecture 3a

Learning the weights of a linear neuron

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

Why the perceptron learning procedure cannot be generalised to hidden layers

- The perceptron convergence procedure works by ensuring that every time the weights change, they get closer to every “generously feasible” set of weights.
 - This type of guarantee cannot be extended to more complex networks in which the average of two good solutions may be a bad solution.
- So “multi-layer” neural networks do not use the perceptron learning procedure.
 - They should never have been called multi-layer perceptrons.

A different way to show that a learning procedure makes progress

- Instead of showing the weights get closer to a good set of weights, show that the actual output values get closer the target values.
 - This can be true even for non-convex problems in which there are many quite different sets of weights that work well and averaging two good sets of weights may give a bad set of weights.
 - It is not true for perceptron learning.
- The simplest example is a linear neuron with a squared error measure.

Linear neurons (also called linear filters)

- The neuron has a real-valued output which is a weighted sum of its inputs
- The aim of learning is to minimize the error summed over all training cases.
 - The error is the squared difference between the desired output and the actual output.

$$y = \sum_i w_i x_i = \mathbf{w}^T \mathbf{x}$$

weight vector
↓
 \mathbf{w}^T
↑
input vector

↑
neuron's estimate of the desired output

Why don't we solve it analytically?

- It is straight-forward to write down a set of equations, one per training case, and to solve for the best set of weights.
 - This is the standard engineering approach so why don't we use it?
- Scientific answer: We want a method that real neurons could use.
- Engineering answer: We want a method that can be generalized to multi-layer, non-linear neural networks.
 - The analytic solution relies on it being linear and having a squared error measure.
 - Iterative methods are usually less efficient but they are much easier to generalize.

A toy example to illustrate the iterative method

- Each day you get lunch at the cafeteria.
 - Your diet consists of fish, chips, and ketchup.
 - You get several portions of each.
- The cashier only tells you the total price of the meal
 - After several days, you should be able to figure out the price of each portion.
- The iterative approach: Start with random guesses for the prices and then adjust them to get a better fit to the observed prices of whole meals.

Solving the equations iteratively

- Each meal price gives a linear constraint on the prices of the portions:

$$price = x_{fish}w_{fish} + x_{chips}w_{chips} + x_{ketchup}w_{ketchup}$$

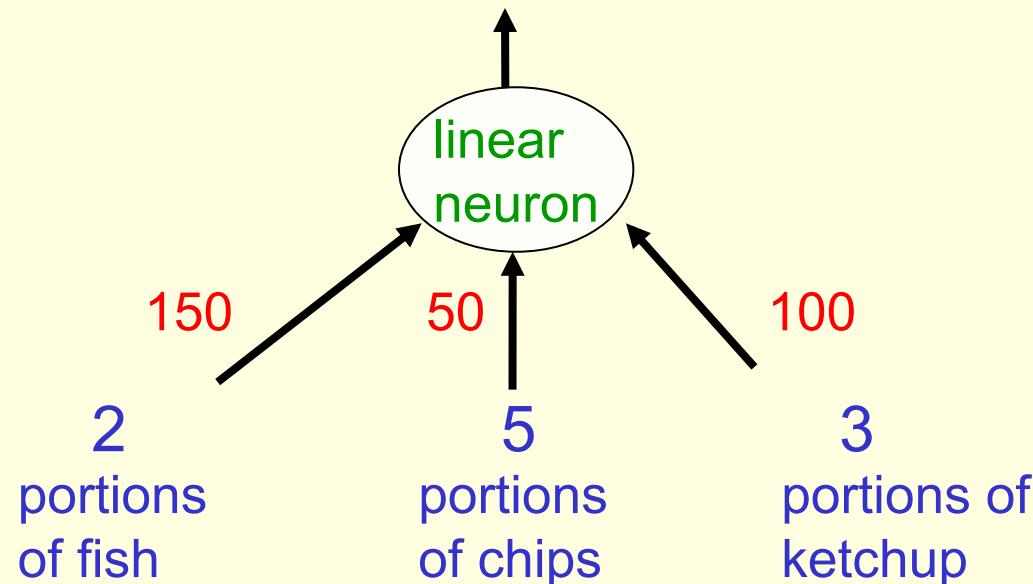
- The prices of the portions are like the weights in of a linear neuron.

$$\mathbf{w} = (w_{fish}, w_{chips}, w_{ketchup})$$

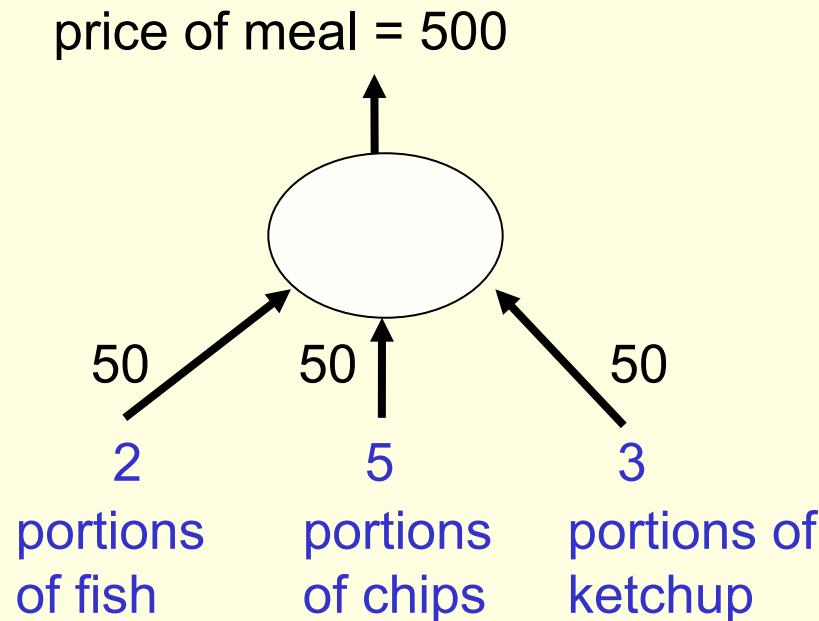
- We will start with guesses for the weights and then adjust the guesses slightly to give a better fit to the prices given by the cashier.

The true weights used by the cashier

Price of meal = 850 = target



A model of the cashier with arbitrary initial weights



- Residual error = 350
- The “delta-rule” for learning is:
$$\Delta w_i = \epsilon x_i (t - y)$$
- With a learning rate ϵ of 1/35, the weight changes are +20, +50, +30
- This gives new weights of 70, 100, 80.
 - Notice that the weight for chips got worse!

Deriving the delta rule

- Define the error as the squared residuals summed over all training cases:

$$E = \frac{1}{2} \sum_{n \in \text{training}} (t^n - y^n)^2$$

- Now differentiate to get error derivatives for weights

$$\frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_n \frac{\partial y^n}{\partial w_i} \frac{dE^n}{dy^n}$$

- The **batch** delta rule changes the weights in proportion to their error derivatives **summed over all training cases**

$$= - \sum_n x_i^n (t^n - y^n)$$

$$\rightarrow \Delta w_i = -\varepsilon \frac{\partial E}{\partial w_i} = \sum_n \varepsilon x_i^n (t^n - y^n)$$

Behaviour of the iterative learning procedure

- Does the learning procedure eventually get the right answer?
 - There may be no perfect answer.
 - By making the learning rate small enough we can get as close as we desire to the best answer.
- How quickly do the weights converge to their correct values?
 - It can be very slow if two input dimensions are highly correlated. If you almost always have the same number of portions of ketchup and chips, it is hard to decide how to divide the price between ketchup and chips.

The relationship between the online delta-rule and the learning rule for perceptrons

- In perceptron learning, we increment or decrement the weight vector by the input vector.
 - But we only change the weights when we make an error.
- In the online version of the delta-rule we increment or decrement the weight vector by the input vector scaled by the residual error and the learning rate.
 - So we have to choose a learning rate. This is annoying.

Neural Networks for Machine Learning

Lecture 3b

The error surface for a linear neuron

Geoffrey Hinton

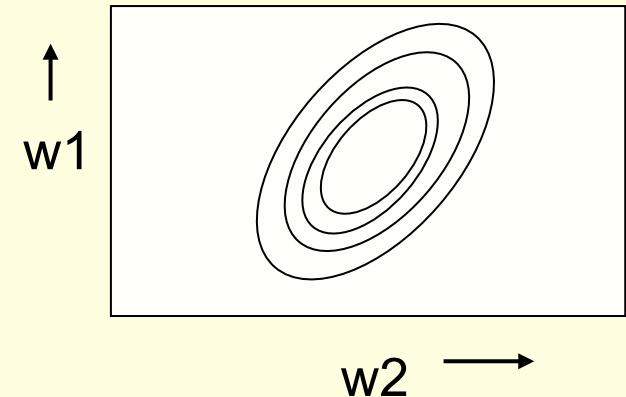
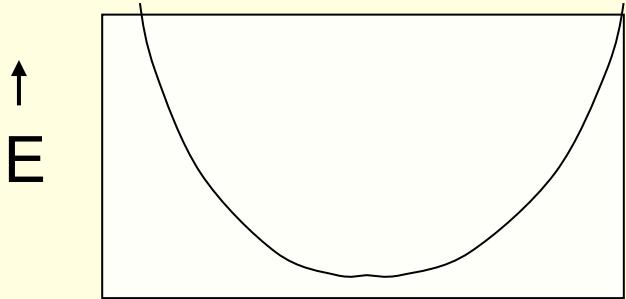
with

Nitish Srivastava

Kevin Swersky

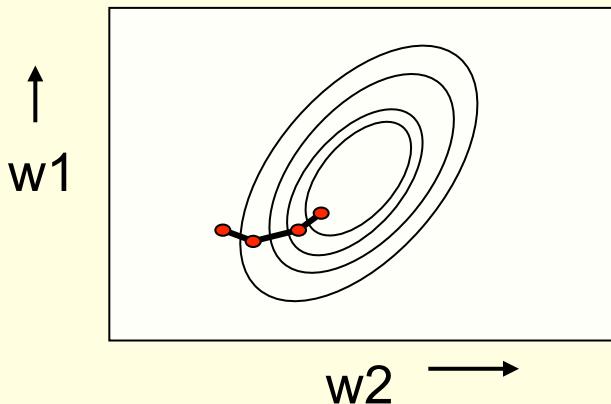
The error surface in extended weight space

- The error surface lies in a space with a horizontal axis for each weight and one vertical axis for the error.
 - For a linear neuron with a squared error, it is a quadratic bowl.
 - Vertical cross-sections are parabolas.
 - Horizontal cross-sections are ellipses.
- For multi-layer, non-linear nets the error surface is much more complicated.



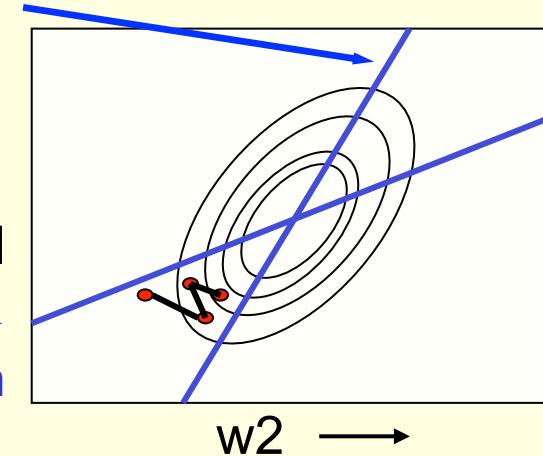
Online versus batch learning

- The simplest kind of batch learning does steepest descent on the error surface.
 - This travels perpendicular to the contour lines.
- The simplest kind of online learning zig-zags around the direction of steepest descent:



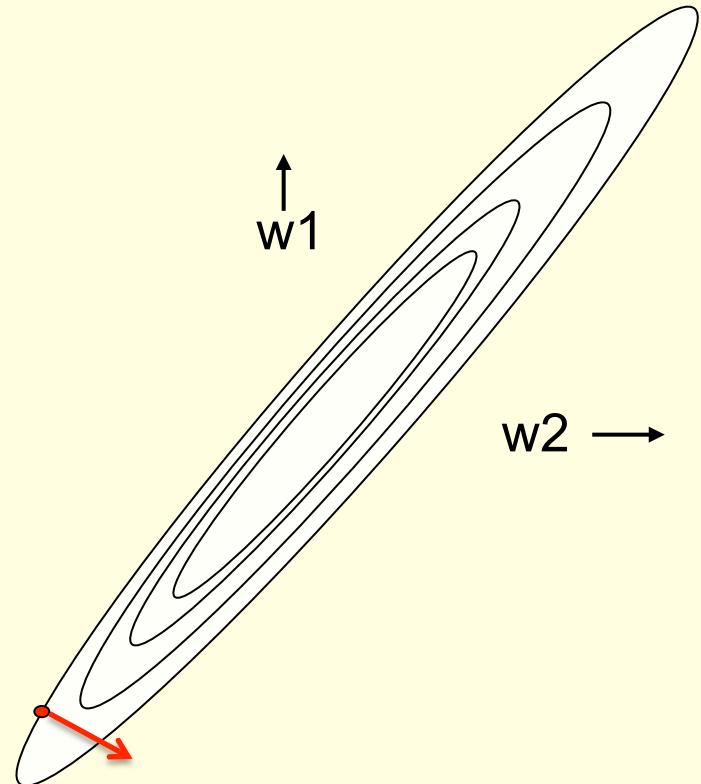
constraint from
training case 1

constraint from
training case 2



Why learning can be slow

- If the ellipse is very elongated, the direction of steepest descent is almost perpendicular to the direction towards the minimum!
 - The red gradient vector has a large component along the short axis of the ellipse and a small component along the long axis of the ellipse.
 - This is just the opposite of what we want.



Neural Networks for Machine Learning

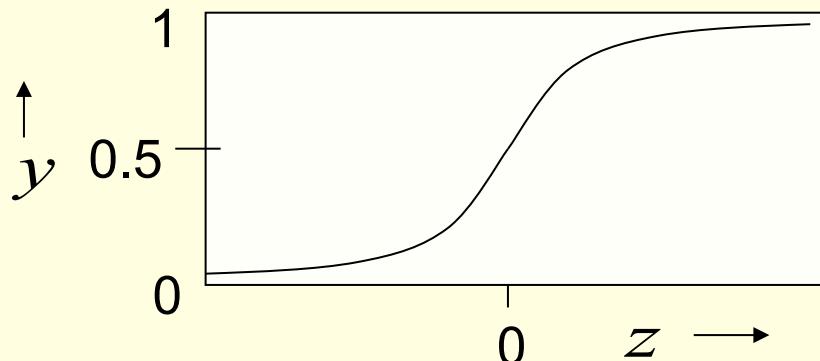
Lecture 3c Learning the weights of a logistic output neuron

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

Logistic neurons

- These give a real-valued output that is a smooth and bounded function of their total input.
 - They have nice derivatives which make learning easy.

$$z = b + \sum_i x_i w_i \qquad y = \frac{1}{1 + e^{-z}}$$



The derivatives of a logistic neuron

- The derivatives of the logit, z , with respect to the inputs and the weights are very simple:

$$z = b + \sum_i x_i w_i$$

$$\frac{\partial z}{\partial w_i} = x_i$$

$$\frac{\partial z}{\partial x_i} = w_i$$

- The derivative of the output with respect to the logit is simple if you express it in terms of the output:

$$y = \frac{1}{1 + e^{-z}}$$

$$\frac{dy}{dz} = y(1 - y)$$

The derivatives of a logistic neuron

$$y = \frac{1}{1 + e^{-z}} = (1 + e^{-z})^{-1}$$

$$\frac{dy}{dz} = \frac{-1(-e^{-z})}{(1 + e^{-z})^2} = \left(\frac{1}{1 + e^{-z}} \right) \left(\frac{e^{-z}}{1 + e^{-z}} \right) = y(1 - y)$$

because $\frac{e^{-z}}{1 + e^{-z}} = \frac{(1 + e^{-z}) - 1}{1 + e^{-z}} = \frac{(1 + e^{-z})}{1 + e^{-z}} \frac{-1}{1 + e^{-z}} = 1 - y$

Using the chain rule to get the derivatives needed for learning the weights of a logistic unit

- To learn the weights we need the derivative of the output with respect to each weight:

$$\frac{\partial y}{\partial w_i} = \frac{\partial z}{\partial w_i} \frac{dy}{dz} = x_i y (1 - y)$$

$$\frac{\partial E}{\partial w_i} = \sum_n \frac{\partial y^n}{\partial w_i} \frac{\partial E}{\partial y^n} = - \sum_n [x_i^n \quad y^n (1 - y^n) \quad (t^n - y^n)]$$

delta-rule

extra term = slope of logistic

Neural Networks for Machine Learning

Lecture 3d The backpropagation algorithm

Geoffrey Hinton

with

Nitish Srivastava

Kevin Swersky

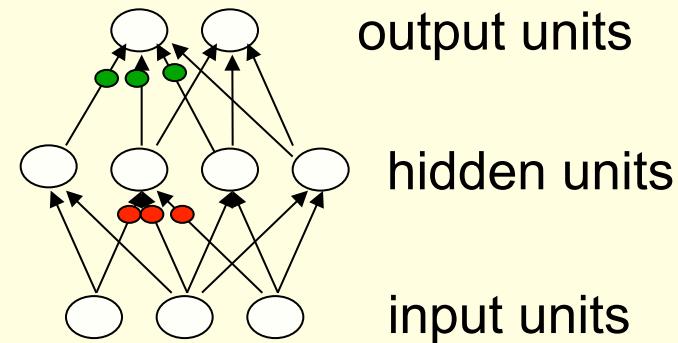
Learning with hidden units (again)

- Networks without hidden units are very limited in the input-output mappings they can model.
- Adding a layer of hand-coded features (as in a perceptron) makes them much more powerful but the hard bit is designing the features.
 - We would like to find good features without requiring insights into the task or repeated trial and error where we guess some features and see how well they work.
- We need to automate the loop of designing features for a particular task and seeing how well they work.

Learning by perturbing weights

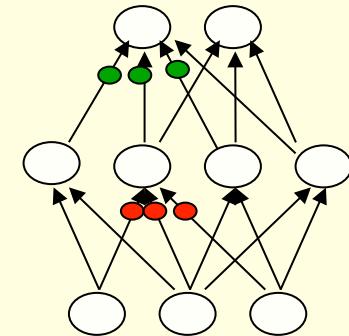
(this idea occurs to everyone who knows about evolution)

- Randomly perturb one weight and see if it improves performance. If so, save the change.
 - This is a form of reinforcement learning.
 - Very inefficient. We need to do multiple forward passes on a representative set of training cases just to change one weight. Backpropagation is much better.
 - Towards the end of learning, large weight perturbations will nearly always make things worse, because the weights need to have the right relative values.



Learning by using perturbations

- We could randomly perturb all the weights in parallel and correlate the performance gain with the weight changes.
 - Not any better because we need lots of trials on each training case to “see” the effect of changing one weight through the noise created by all the changes to other weights.
- A better idea: Randomly perturb the activities of the hidden units.
 - Once we know how we want a hidden activity to change on a given training case, we can **compute** how to change the weights.
 - There are fewer activities than weights, but backpropagation still wins by a factor of the number of neurons.



The idea behind backpropagation

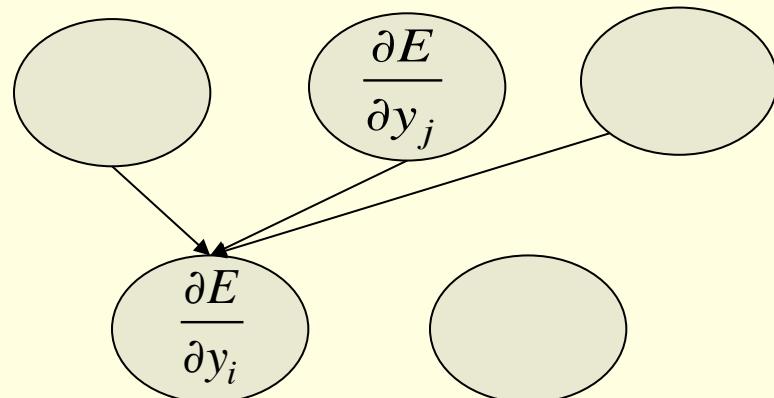
- We don't know what the hidden units ought to do, but we can compute how fast the error changes as we change a hidden activity.
 - Instead of using desired activities to train the hidden units, use **error derivatives w.r.t. hidden activities**.
 - Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined.
- We can compute error derivatives for all the hidden units efficiently at the same time.
 - Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit.

Sketch of the backpropagation algorithm on a single case

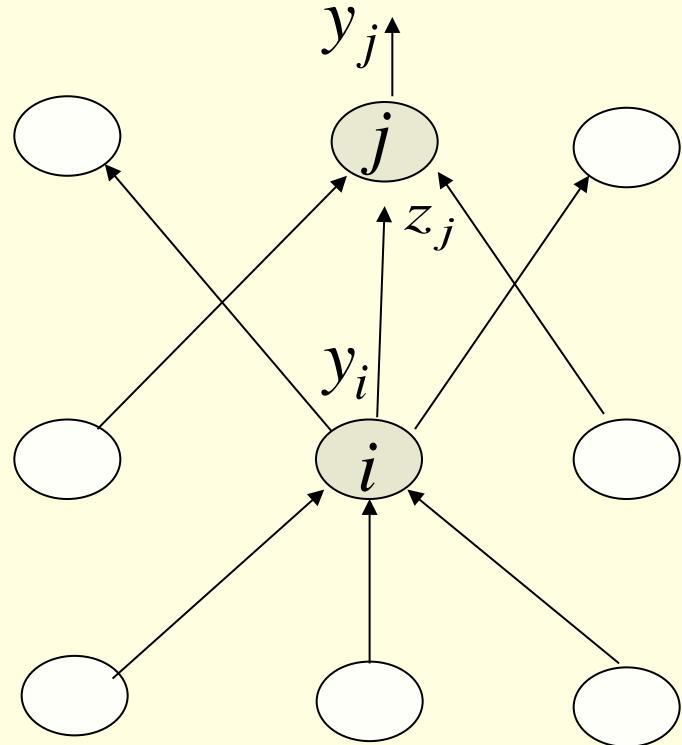
- First convert the discrepancy between each output and its target value into an error derivative.
- Then compute error derivatives in each hidden layer from error derivatives in the layer above.
- Then use error derivatives *w.r.t.* activities to get error derivatives *w.r.t.* the incoming weights.

$$E = \frac{1}{2} \sum_{j \in \text{output}} (t_j - y_j)^2$$

$$\frac{\partial E}{\partial y_j} = -(t_j - y_j)$$



Backpropagating dE/dy



$$\frac{\partial E}{\partial z_j} = \frac{dy_j}{dz_j} \frac{\partial E}{\partial y_j} = y_j (1 - y_j) \frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{dz_j}{dy_i} \frac{\partial E}{\partial z_j} = \sum_j w_{ij} \frac{\partial E}{\partial z_j}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}} \frac{\partial E}{\partial z_j} = y_i \frac{\partial E}{\partial z_j}$$

Neural Networks for Machine Learning

Lecture 3e

How to use the derivatives computed by the
backpropagation algorithm

Geoffrey Hinton

with

Nitish Srivastava

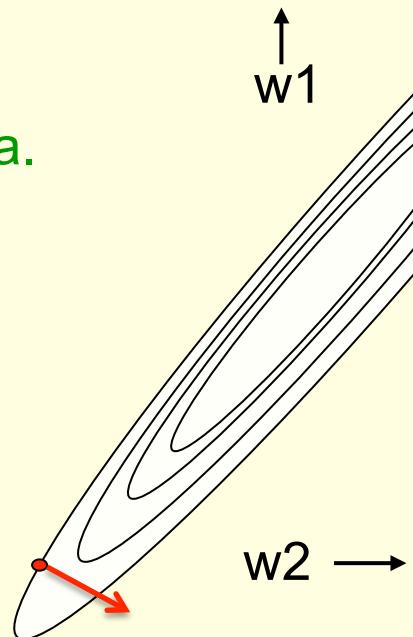
Kevin Swersky

Converting error derivatives into a learning procedure

- The backpropagation algorithm is an efficient way of computing the error derivative dE/dw for every weight on a single training case.
- To get a fully specified learning procedure, we still need to make a lot of other decisions about how to use these error derivatives:
 - Optimization issues: How do we use the error derivatives on individual cases to discover a good set of weights? (lecture 6)
 - Generalization issues: How do we ensure that the learned weights work well for cases we did not see during training? (lecture 7)
- We now have a very brief overview of these two sets of issues.

Optimization issues in using the weight derivatives

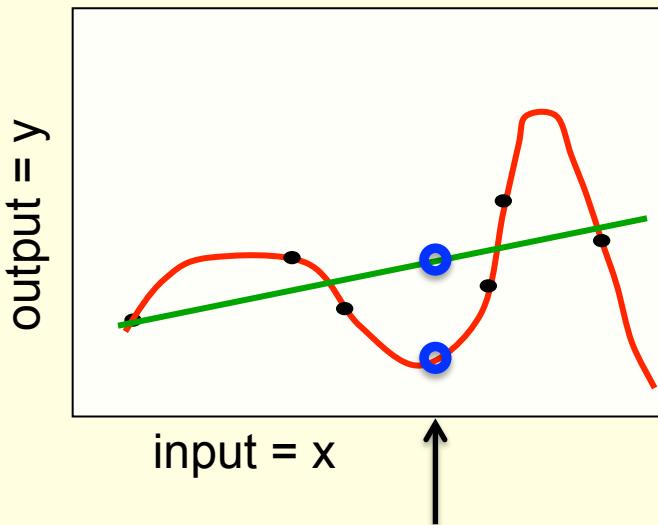
- How often to update the weights
 - Online: after each training case.
 - Full batch: after a full sweep through the training data.
 - Mini-batch: after a small sample of training cases.
- How much to update (discussed further in lecture 6)
 - Use a fixed learning rate?
 - Adapt the global learning rate?
 - Adapt the learning rate on each connection separately?
 - Don't use steepest descent?



Overfitting: The downside of using powerful models

- The training data contains information about the regularities in the mapping from input to output. But it also contains two types of noise.
 - The target values may be unreliable (usually only a minor worry).
 - There is **sampling error**. There will be accidental regularities just because of the particular training cases that were chosen.
- When we fit the model, it cannot tell which regularities are real and which are caused by sampling error.
 - So it fits both kinds of regularity.
 - If the model is very flexible it can model the sampling error really well. **This is a disaster.**

A simple example of overfitting



- Which model do you trust?
 - The complicated model fits the data better.
 - But it is not economical.
- A model is convincing when it fits a lot of data surprisingly well.
 - It is not surprising that a complicated model can fit a small amount of data well.

Ways to reduce overfitting

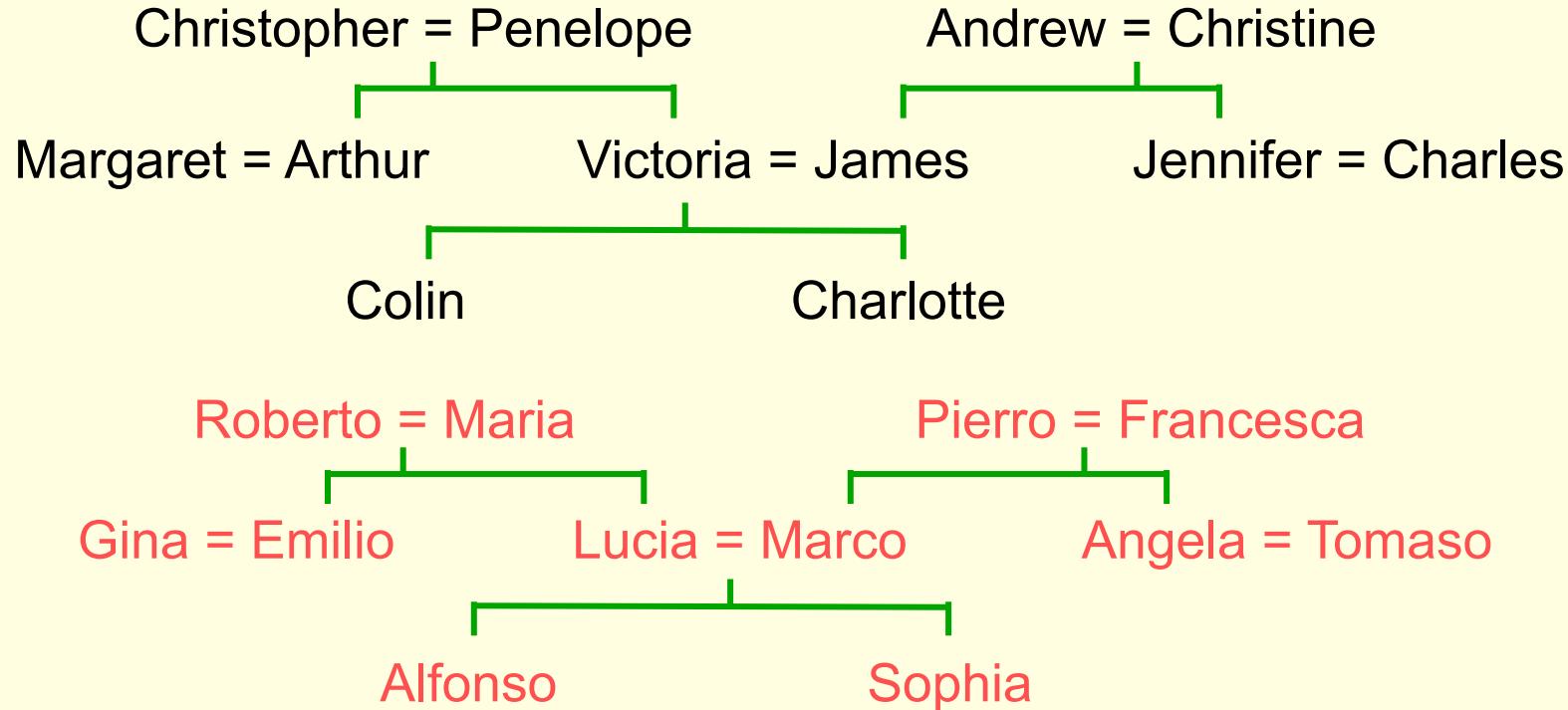
- A large number of different methods have been developed.
 - Weight-decay
 - Weight-sharing
 - Early stopping
 - Model averaging
 - Bayesian fitting of neural nets
 - Dropout
 - Generative pre-training
- Many of these methods will be described in lecture 7.

Neural Networks for Machine Learning

Lecture 4a Learning to predict the next word

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

A simple example of relational information



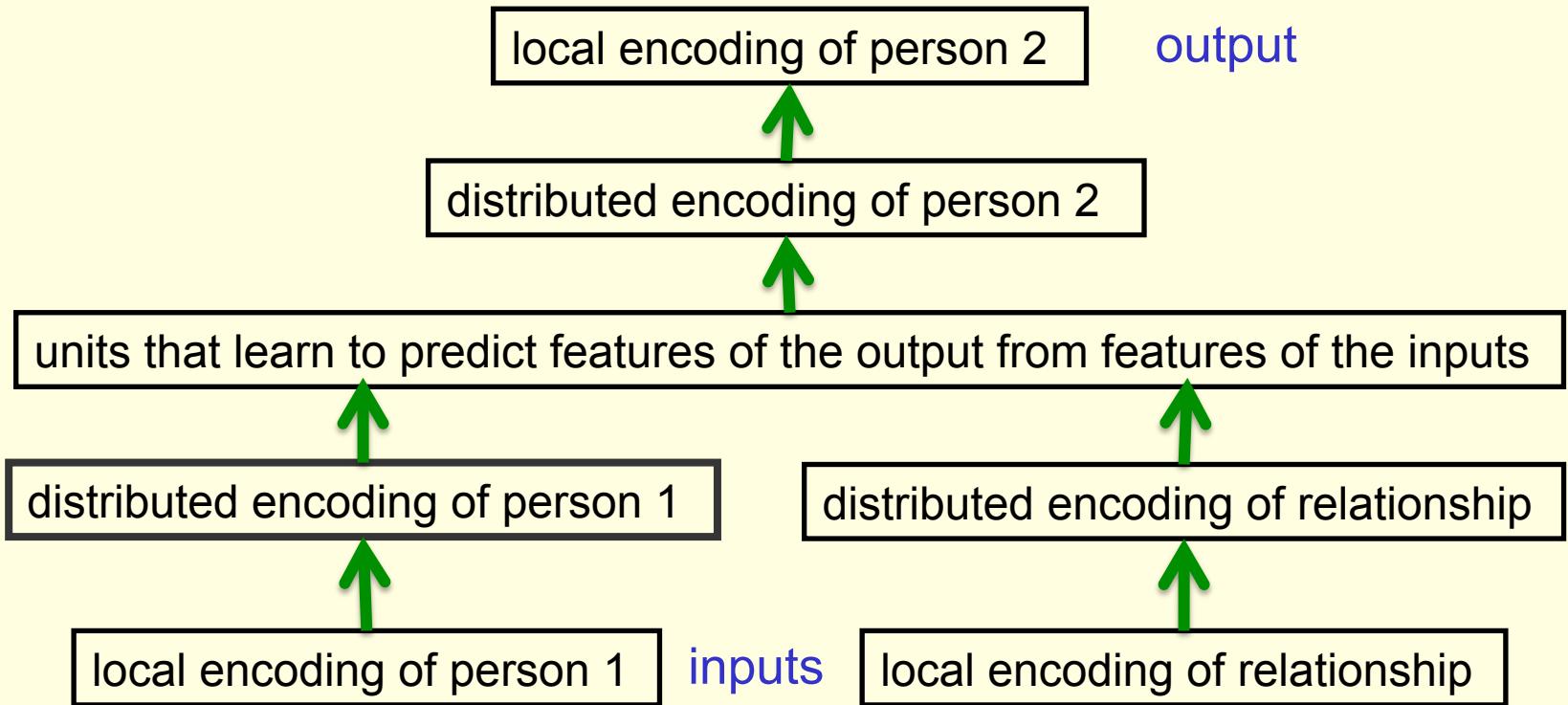
Another way to express the same information

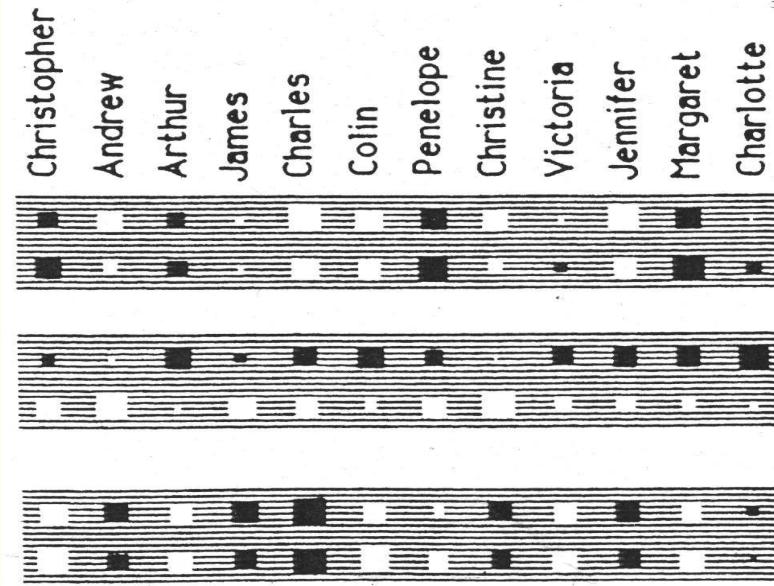
- Make a set of propositions using the 12 relationships:
 - son, daughter, nephew, niece, father, mother, uncle, aunt
 - brother, sister, husband, wife
- (colin has-father james)
- (colin has-mother victoria)
- (james has-wife victoria) this follows from the two above
- (charlotte has-brother colin)
- (victoria has-brother arthur)
- (charlotte has-uncle arthur) this follows from the above

A relational learning task

- Given a large set of triples that come from some family trees, figure out the regularities.
 - The obvious way to express the regularities is as symbolic rules
 $(x \text{ has-mother } y) \& (y \text{ has-husband } z) \Rightarrow (x \text{ has-father } z)$
- Finding the symbolic rules involves a difficult search through a very large discrete space of possibilities.
- Can a neural network capture the same knowledge by searching through a continuous space of weights?

The structure of the neural net





Christopher = Penelope
Margaret = Arthur

Andrew = Christine
Victoria = James
Colin
Charlotte

Jennifer = Charles

What the network learns

- The six hidden units in the bottleneck connected to the input representation of person 1 learn to represent features of people that are useful for predicting the answer.
 - Nationality, generation, branch of the family tree.
- These features are only useful if the other bottlenecks use similar representations and the central layer learns how features predict other features. For example:

Input person is of generation 3 **and**

relationship requires answer to be one generation up
implies

Output person is of generation 2

Another way to see that it works

- Train the network on all but 4 of the triples that can be made using the 12 relationships
 - It needs to sweep through the training set many times adjusting the weights slightly each time.
- Then test it on the 4 held-out cases.
 - It gets about 3/4 correct.
 - This is good for a 24-way choice.
 - On much bigger datasets we can train on a much smaller fraction of the data.

A large-scale example

- Suppose we have a database of millions of relational facts of the form $(A \text{ } R \text{ } B)$.
 - We could train a net to discover feature vector representations of the terms that allow the third term to be predicted from the first two.
 - Then we could use the trained net to find very unlikely triples. These are good candidates for errors in the database.
- Instead of predicting the third term, we could use all three terms as input and predict the probability that the fact is correct.
 - To train such a net we need a good source of false facts.

Neural Networks for Machine Learning

Lecture 4b A brief diversion into cognitive science

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

What the family trees example tells us about concepts

- There has been a long debate in cognitive science between two rival theories of what it means to have a concept:

The feature theory: A concept is a set of semantic features.

- This is good for explaining similarities between concepts.
- Its convenient: a concept is a vector of feature activities.

The structuralist theory: The meaning of a concept lies in its relationships to other concepts.

- So conceptual knowledge is best expressed as a relational graph.
- Minsky used the limitations of perceptrons as evidence against feature vectors and in favor of relational graph representations.

Both sides are wrong

- These two theories need not be rivals. A neural net can use vectors of semantic features to implement a relational graph.
 - In the neural network that learns family trees, no explicit inference is required to arrive at the intuitively obvious consequences of the facts that have been explicitly learned.
 - The net can “intuit” the answer in a forward pass.
- We may use explicit rules for conscious, deliberate reasoning, but we do a lot of commonsense, analogical reasoning by just “seeing” the answer with no conscious intervening steps.
 - Even when we are using explicit rules, we need to just see which rules to apply.

Localist and distributed representations of concepts

- The obvious way to implement a relational graph in a neural net is to treat a neuron as a node in the graph and a connection as a binary relationship. But this “localist” method will not work:
 - We need many different types of relationship and the connections in a neural net do not have discrete labels.
 - We need ternary relationships as well as binary ones.
e.g. A is between B and C.
- The right way to implement relational knowledge in a neural net is still an open issue.
 - But many neurons are probably used for each concept and each neuron is probably involved in many concepts. This is called a “distributed representation”.

Neural Networks for Machine Learning

Lecture 4c

Another diversion: The softmax output function

Geoffrey Hinton

with

Nitish Srivastava

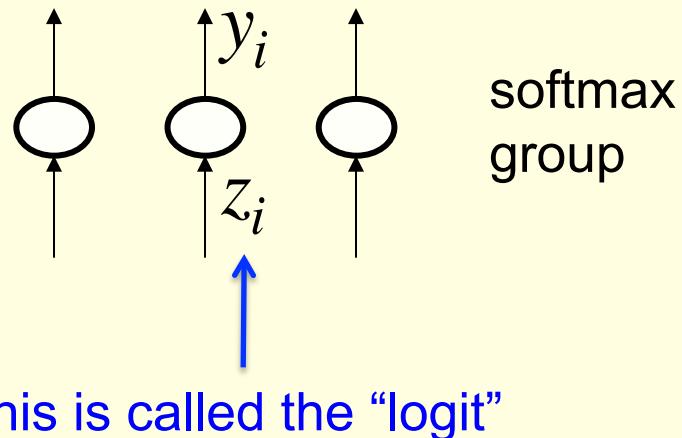
Kevin Swersky

Problems with squared error

- The squared error measure has some drawbacks:
 - If the desired output is 1 and the actual output is 0.00000001 there is almost no gradient for a logistic unit to fix up the error.
 - If we are trying to assign probabilities to mutually exclusive class labels, we know that the outputs should sum to 1, but we are depriving the network of this knowledge.
- Is there a different cost function that works better?
 - Yes: Force the outputs to represent a probability distribution across discrete alternatives.

Softmax

The output units in a softmax group use a non-local non-linearity:



$$y_i = \frac{e^{z_i}}{\sum_{j \in group} e^{z_j}}$$

$$\frac{\partial y_i}{\partial z_i} = y_i (1 - y_i)$$

Cross-entropy: the right cost function to use with softmax

- The right cost function is the negative log probability of the right answer.
- C has a very big gradient when the target value is 1 and the output is almost zero.
 - A value of 0.000001 is much better than 0.00000001
 - The steepness of dC/dy exactly balances the flatness of dy/dz

$$C = - \sum_j t_j \log y_j$$


target value

$$\frac{\partial C}{\partial z_i} = \sum_j \frac{\partial C}{\partial y_j} \frac{\partial y_j}{\partial z_i} = y_i - t_i$$

Neural Networks for Machine Learning

Lecture 4d Neuro-probabilistic language models

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

A basic problem in speech recognition

- We cannot identify phonemes perfectly in noisy speech
 - The acoustic input is often ambiguous: there are several different words that fit the acoustic signal equally well.
- People use their understanding of the meaning of the utterance to hear the right words.
 - We do this unconsciously when we wreck a nice beach.
 - We are very good at it.
- This means speech recognizers have to know which words are likely to come next and which are not.
 - Fortunately, words can be predicted quite well without full understanding.

The standard “trigram” method

- Take a huge amount of text and count the frequencies of all triples of words.
- Use these frequencies to make bets on the relative probabilities of words given the previous two words:

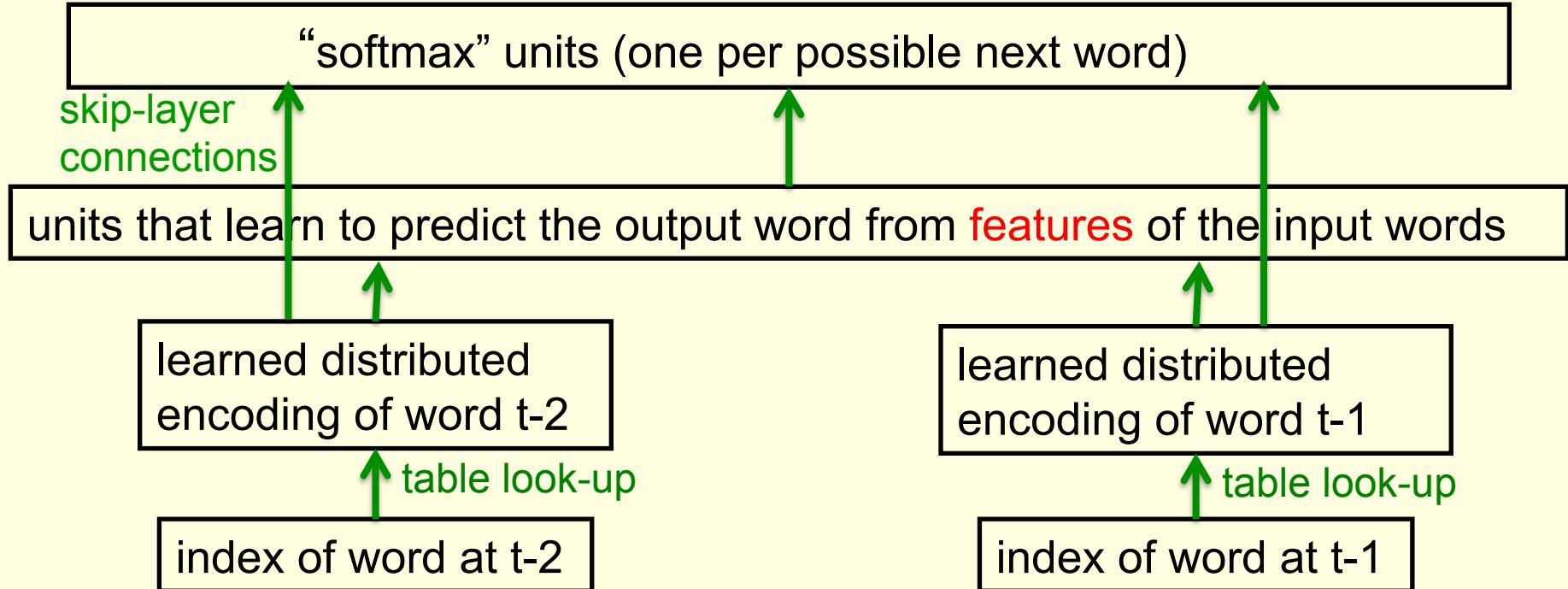
$$\frac{p(w_3 = c \mid w_2 = b, w_1 = a)}{p(w_3 = d \mid w_2 = b, w_1 = a)} = \frac{\text{count}(abc)}{\text{count}(abd)}$$

- Until very recently this was the state-of-the-art.
 - We cannot use a much bigger context because there are too many possibilities to store and the counts would mostly be zero.
 - We have to “back-off” to digrams when the count for a trigram is too small.
 - The probability is not zero just because the count is zero!

Information that the trigram model fails to use

- Suppose we have seen the sentence
“the cat got squashed in the garden on friday”
- This should help us predict words in the sentence
“the dog got flattened in the yard on monday”
- A trigram model does not understand the similarities between
 - cat/dog squashed/flattened garden/yard friday/monday
- To overcome this limitation, we need to use the semantic and syntactic features of previous words to predict the features of the next word.
 - Using a feature representation also allows a context that contains many more previous words (e.g. 10).

Bengio's neural net for predicting the next word



A problem with having 100,000 output words

- Each unit in the last hidden layer has 100,000 outgoing weights.
 - So we cannot afford to have many hidden units.
 - Unless we have a huge number of training cases.
 - We could make the last hidden layer small, but then its hard to get the 100,000 probabilities right.
 - The small probabilities are often relevant.
- Is there a better way to deal with such a large number of outputs?

Neural Networks for Machine Learning

Lecture 4e

Ways to deal with the large number of possible outputs in neuro-probabilistic language models

Geoffrey Hinton

with

Nitish Srivastava

Kevin Swersky

A serial architecture

Try all candidate next words one at a time.

logit score for the candidate word

This allows the learned feature vector representation to be used for the candidate word.

hidden units that discover good or bad combinations of features

learned distributed encoding of word t-2

learned distributed encoding of word t-1

learned distributed encoding of **candidate**

index of word at t-2

index of word at t-1

index of candidate

table look-up

table look-up

table look-up

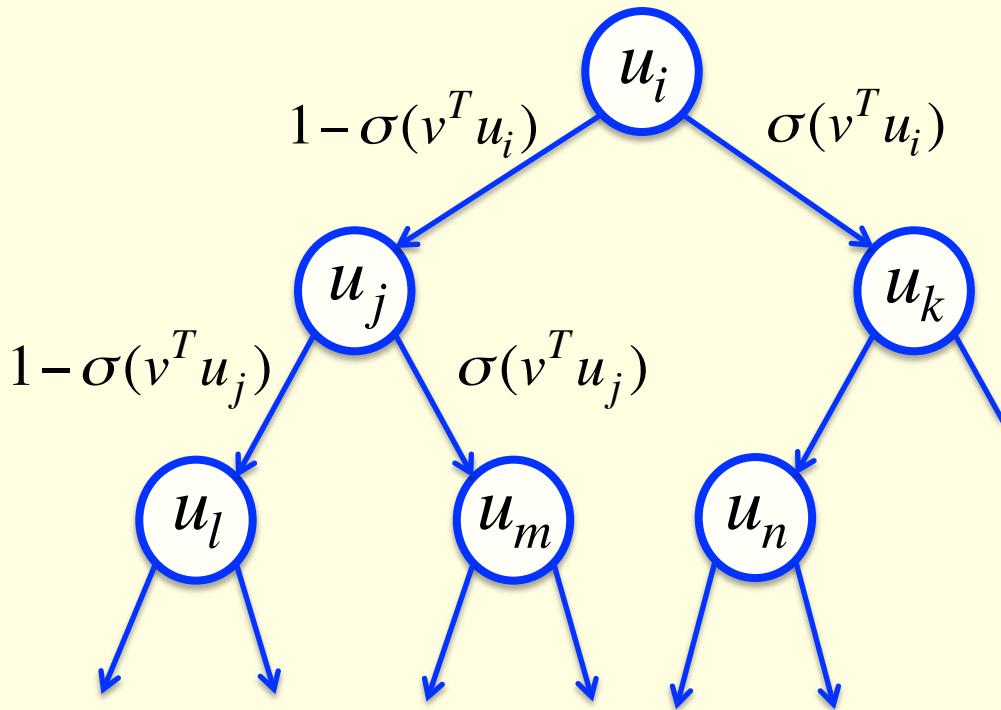


Learning in the serial architecture

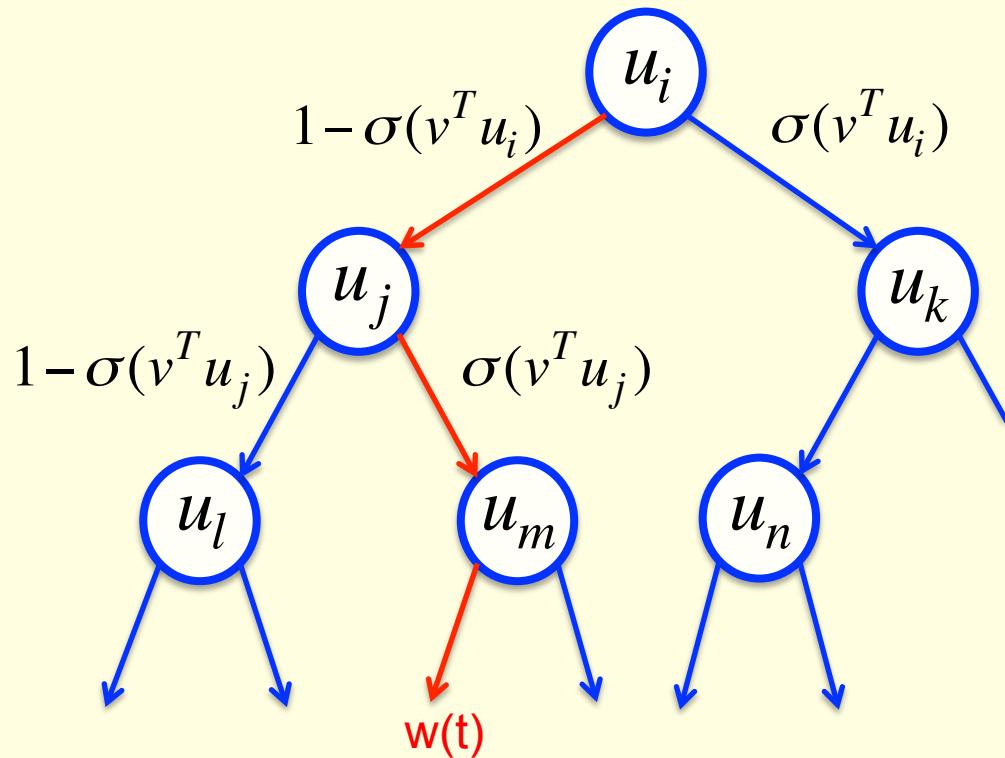
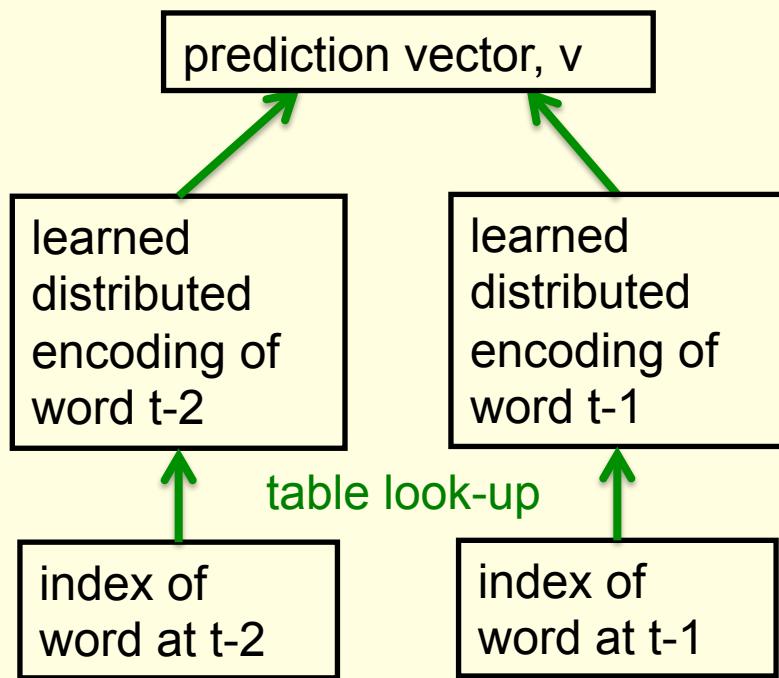
- After computing the logit score for each candidate word, use all of the logits in a softmax to get word probabilities.
- The difference between the word probabilities and their target probabilities gives cross-entropy error derivatives.
 - The derivatives try to raise the score of the correct candidate and lower the scores of its high-scoring rivals.
- We can save a lot of time if we only use a small set of candidates suggested by some other kind of predictor.
 - For example, we could use the neural net to revise the probabilities of the words that the trigram model thinks are likely.

Learning to predict the next word by predicting a path through a tree (Minih and Hinton, 2009)

- Arrange all the words in a binary tree with words as the leaves.
- Use the previous context to generate a “prediction vector”, v.
 - Compare v with a learned vector, u, at each node of the tree.
 - Apply the logistic function to the scalar product of u and v to predict the probabilities of taking the two branches of the tree.



A picture of the learning



A convenient decomposition

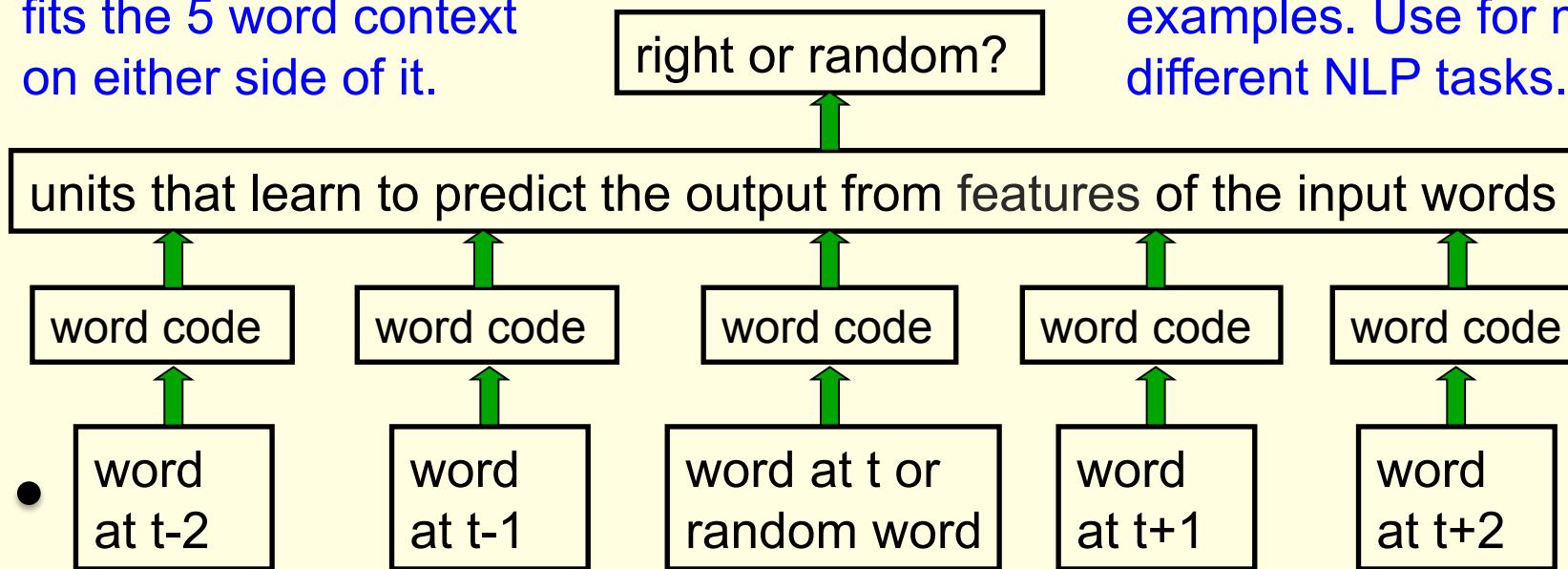
- Maximizing the log probability of picking the target word is equivalent to maximizing the sum of the log probabilities of taking all the branches on the path that leads to the target word.
 - So during learning, we only need to consider the nodes on the correct path. This is an exponential win: $\log(N)$ instead of N .
 - For each of these nodes, we know the correct branch and we know the current probability of taking it so we can get derivatives for learning both the prediction vector v and that node vector u .
- Unfortunately, it is still slow at test time.

A simpler way to learn feature vectors for words

(Collobert and Weston, 2008)

Learn to judge if a word
fits the 5 word context
on either side of it.

Train on ~600 million
examples. Use for many
different NLP tasks.



Displaying the learned feature vectors in a 2-D map

- We can get an idea of the quality of the learned feature vectors by displaying them in a 2-D map.
 - Display very similar vectors very close to each other.
 - Use a multi-scale method called “t-sne” that also displays similar clusters near each other.
- The learned feature vectors capture lots of subtle semantic distinctions, just by looking at strings of words.
 - No extra supervision is required.
 - The information is all in the contexts that the word is used in.
 - Consider “She scrommed him with the frying pan.”

Part of a 2-D map of the 2500 most common words

A 2D map showing the relationships between 2500 common words. The words are represented as nodes in a network, with their positions indicating semantic similarity. The words are clustered into several main groups:

- Sports and Competitions:** winner, player, team, club, sport, league, stadium, tournament, championships, finals, olympics, cup, bowl, medal, prize, award, awards.
- Leagues and Events:** nfl, olympic, champion, olympics.
- Participants and Fans:** rubber, Hockey, soccer, basketball, baseball, wrestling, sports, matches, games, clubs, teams, players, fans.

The words are arranged in a roughly circular pattern, with more specific terms like "nfl" and "olympic" at the center and more general or descriptive terms like "winner" and "player" towards the periphery.

virginia
columbia missouri
indiana maryland
colorado tennessee
washington oregon kansas carolina
california minnesota
houston philadelphia pennsylvania
detroit toronto ontario massachusetts
hollywood sydney melbourne montreal manchester
london victoria
berlin quebec
moscow mexico scotland
wales
canada ireland britain
australia sweden
singapore america norway spain
europe austria
asia germany poland
africa russia
india japan rome
korea china egypt
pakistam vietnam israel
iran

rather increasingly further
otherwise later

entirely completely
newly greatly

farmland

heavily quickly fully
well closely briefly ever
widely directly already

only just both
even either
then once yet

frequently officially
specifically regularly initially
simply originally usually

so

largely currently soon shortly
mainly also n't never immediately
mostly still not generally eventually again

especially formerly typically apparently
occasionally intimately

sometimes finally twice
notably instead meanwhile
likely probably thus therefore
possibly however
perhaps hence

none

afterwards here
ago today there

which
that
whom
what
how
whether
why

nor
but
as
if
where
because when
though although
whilst
before
except

Neural Networks for Machine Learning

Lecture 5a

Why object recognition is difficult

Geoffrey Hinton

with

Nitish Srivastava

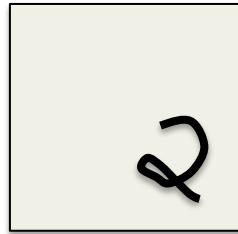
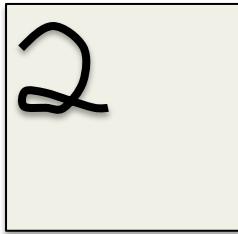
Kevin Swersky

Things that make it hard to recognize objects

- **Segmentation:** Real scenes are cluttered with other objects:
 - Its hard to tell which pieces go together as parts of the same object.
 - Parts of an object can be hidden behind other objects.
- **Lighting:** The intensities of the pixels are determined as much by the lighting as by the objects.
- **Deformation:** Objects can deform in a variety of non-affine ways:
 - e.g a hand-written 2 can have a large loop or just a cusp.
- **Affordances:** Object classes are often defined by how they are used:
 - Chairs are things designed for sitting on so they have a wide variety of physical shapes.

More things that make it hard to recognize objects

- **Viewpoint:** Changes in viewpoint cause changes in images that standard learning methods cannot cope with.
 - Information hops between input dimensions (*i.e.* pixels)
- Imagine a medical database in which the age of a patient sometimes hops to the input dimension that normally codes for weight!
 - To apply machine learning we would first want to eliminate this dimension-hopping.



Neural Networks for Machine Learning

Lecture 5b Ways to achieve viewpoint invariance

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

Some ways to achieve viewpoint invariance

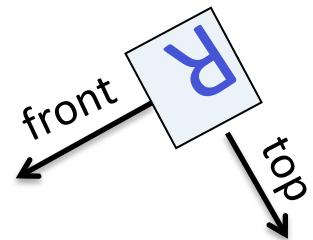
- We are so good at viewpoint invariance that it is hard to appreciate how difficult it is.
 - Its one of the main difficulties in making computers perceive.
 - We still don't have generally accepted solutions.
- There are several different approaches:
 - Use redundant invariant features.
 - Put a box around the object and use normalized pixels.
 - Lecture 5c: Use replicated features with pooling. This is called “convolutional neural nets”
 - Use a hierarchy of parts that have explicit poses relative to the camera (this will be described in detail later in the course).

The invariant feature approach

- Extract a large, redundant set of features that are invariant under transformations
 - e.g. pair of roughly parallel lines with a red dot between them.
A diagram consisting of two thin, slightly curved black lines that are roughly parallel to each other. A small red dot is positioned exactly halfway between the two lines, representing a feature extracted from an image.
 - This is what baby herring gulls use to know where to peck for food.
- With enough invariant features, there is only one way to assemble them into an object.
 - We don't need to represent the relationships between features directly because they are captured by other features.
- But for recognition, we must avoid forming features from parts of different objects.

The judicious normalization approach

- Put a box around the object and use it as a coordinate frame for a set of normalized pixels.
 - This solves the dimension-hopping problem. If we choose the box correctly, the same part of an object always occurs on the same normalized pixels.
 - The box can provide invariance to many degrees of freedom: **translation, rotation, scale, shear, stretch ...**
- But choosing the box is difficult because of:
 - Segmentation errors, occlusion, unusual orientations.
- We need to recognize the shape to get the box right!



We recognize this letter before we do mental rotation to decide if it's a mirror image.

The brute force normalization approach

- When training the recognizer, use well-segmented, upright images to fit the correct box.
- At test time try all possible boxes in a range of positions and scales.
 - This approach is widely used for detecting upright things like faces and house numbers in unsegmented images.
 - It is much more efficient if the recognizer can cope with some variation in position and scale so that we can use a coarse grid when trying all possible boxes.

Neural Networks for Machine Learning

Lecture 5c

Convolutional neural networks for hand-written digit recognition

Geoffrey Hinton

with

Nitish Srivastava

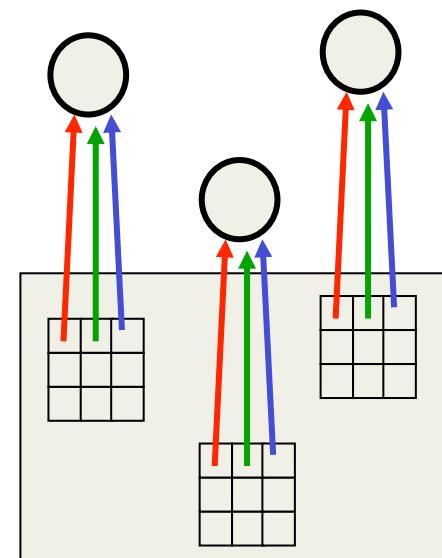
Kevin Swersky

The replicated feature approach

(currently the dominant approach for neural networks)

- Use many different copies of the same feature detector with different positions.
 - Could also replicate across scale and orientation (tricky and expensive)
 - Replication greatly reduces the number of free parameters to be learned.
- Use several different feature types, each with its own map of replicated detectors.
 - Allows each patch of image to be represented in several ways.

The red connections all have the same weight.



Backpropagation with weight constraints

- It's easy to modify the backpropagation algorithm to incorporate linear constraints between the weights.
- We compute the gradients as usual, and then modify the gradients so that they satisfy the constraints.
 - So if the weights started off satisfying the constraints, they will continue to satisfy them.

To constrain: $w_1 = w_2$
we need: $\Delta w_1 = \Delta w_2$

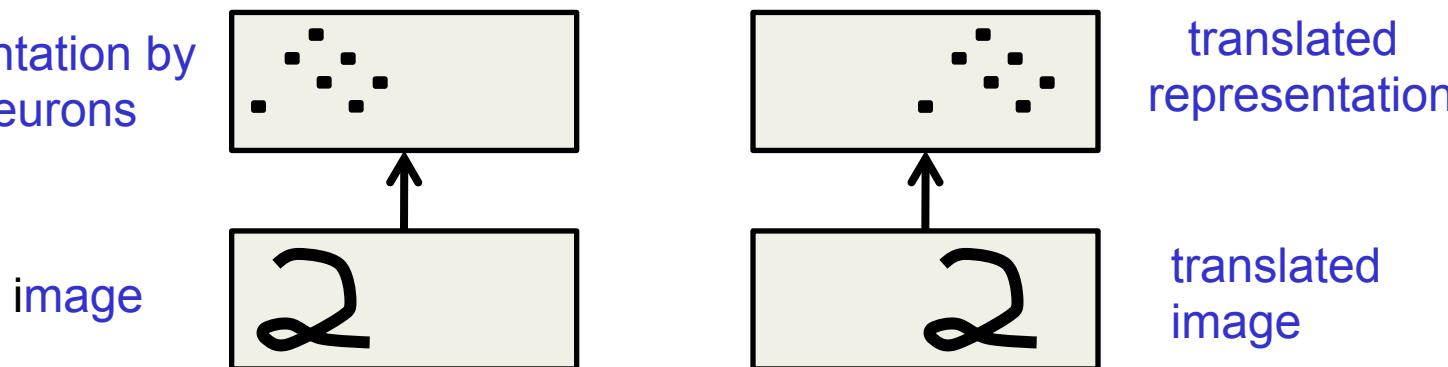
compute: $\frac{\partial E}{\partial w_1}$ and $\frac{\partial E}{\partial w_2}$

use $\frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2}$ *for* w_1 *and* w_2

What does replicating the feature detectors achieve?

- **Equivariant activities:** Replicated features do **not** make the neural activities invariant to translation. The activities are equivariant.

representation by active neurons



- **Invariant knowledge:** If a feature is useful in some locations during training, detectors for that feature will be available in all locations during testing.

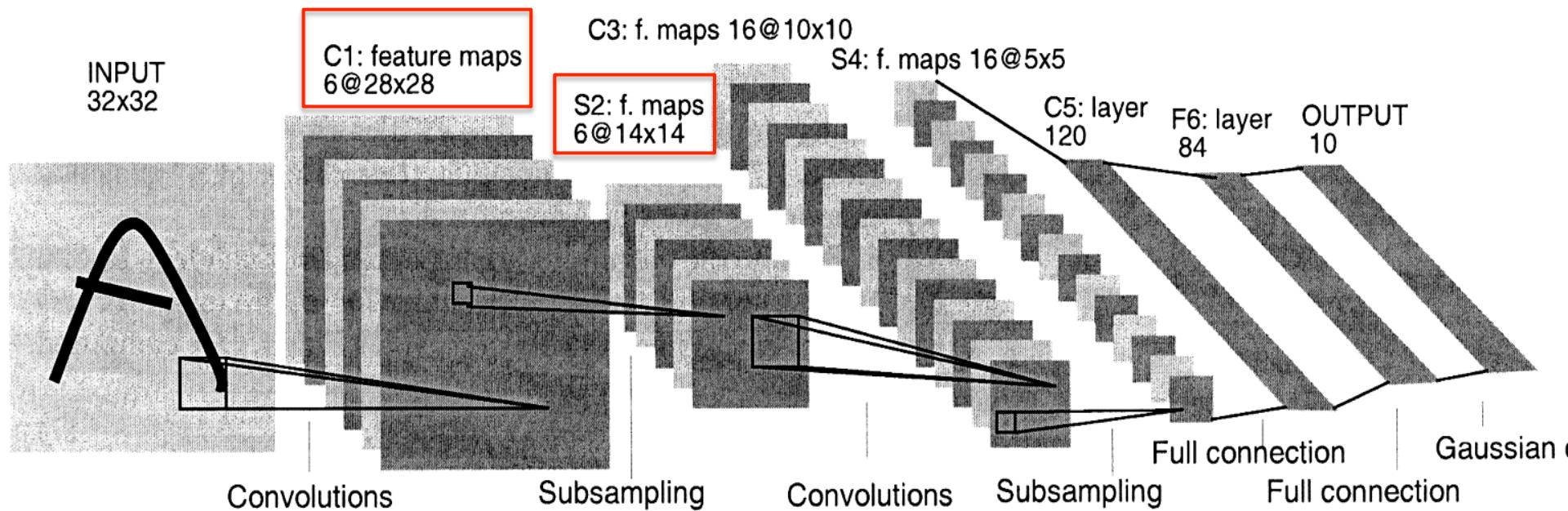
Pooling the outputs of replicated feature detectors

- Get a small amount of translational invariance at each level by averaging four neighboring replicated detectors to give a single output to the next level.
 - This reduces the number of inputs to the next layer of feature extraction, thus allowing us to have many more different feature maps.
 - Taking the maximum of the four works slightly better.
- **Problem:** After several levels of pooling, we have lost information about the precise positions of things.
 - This makes it impossible to use the precise spatial relationships between high-level parts for recognition.

Le Net

- Yann LeCun and his collaborators developed a really good recognizer for handwritten digits by using backpropagation in a feedforward net with:
 - Many hidden layers
 - Many maps of replicated units in each layer.
 - Pooling of the outputs of nearby replicated units.
 - A wide net that can cope with several characters at once even if they overlap.
 - A clever way of training a complete system, not just a recognizer.
- This net was used for reading ~10% of the checks in North America.
- Look the impressive demos of LENET at <http://yann.lecun.com>

The architecture of LeNet5



4	3	2	1	5	8	2	3	6	1
4->6	3->5	8->2	2->1	5->3	4->8	2->8	3->5	6->5	7->3
9	8	7	5	7	6	7	3	3	4
9->4	8->0	7->8	5->3	8->7	0->6	3->7	2->7	8->3	9->4
8	3	4	3	0	9	9	1	4	1
8->2	5->3	4->8	3->9	6->0	9->8	4->9	6->1	9->4	9->1
9	0	1	3	3	9	6	0	2	6
9->4	2->0	6->1	3->5	3->2	9->5	6->0	6->0	6->0	6->8
4	7	9	4	2	7	4	9	9	9
4->6	7->3	9->4	4->6	2->7	9->7	4->3	9->4	9->4	9->4
7	4	8	3	8	6	8	3	3	9
8->7	4->2	8->4	3->5	8->4	6->5	8->5	3->8	3->8	9->8
1	9	6	0	6	7	0	1	4	1
1->5	9->8	6->3	0->2	6->5	9->5	0->7	1->6	4->9	2->1
2	8	4	2	7	1	9	1	6	5
2->8	8->5	4->9	7->2	7->2	6->5	9->7	6->1	5->6	5->0
1	2								
4->9	2->8								

The 82 errors made by LeNet5

Notice that most of the errors are cases that people find quite easy.

The human error rate is probably 20 to 30 errors but nobody has had the patience to measure it.

Priors and Prejudice

- We can put our prior knowledge about the task into the network by designing appropriate:
 - Connectivity.
 - Weight constraints.
 - Neuron activation functions
- This is less intrusive than hand-designing the features.
 - But it still prejudices the network towards the particular way of solving the problem that we had in mind.
- Alternatively, we can use our prior knowledge to create a whole lot more training data.
 - This may require a lot of work (Hofman&Tresp, 1993)
 - It may make learning take much longer.
- It allows optimization to discover clever ways of using the multi-layer network that we did not think of.
 - And we may never fully understand how it does it.

The brute force approach

- LeNet uses knowledge about the invariances to **design**:
 - the local connectivity
 - the weight-sharing
 - the pooling.
- This achieves about 80 errors.
 - This can be reduced to about 40 errors by using many different transformations of the input and other tricks (Ranzato 2008)
- Ciresan *et. al.* (2010) inject knowledge of invariances by creating a huge amount of carefully designed extra training data:
 - For each training image, they produce many new training examples by applying many different transformations.
 - They can then train a large, deep, dumb net on a GPU without much overfitting.
- They achieve about 35 errors.

The errors made by the Ciresan et. al. net

1 2 1 7	1 1 7 1	9 8 9 8	9 9 5 9	9 9 7 9	2 5 3 5	3 8 2 3
4 9 4 9	5 5 3 5	9 4 9 7	4 9 4 9	4 4 9 4	2 2 0 2	5 5 3 5
6 6 1 6	4 4 9 4	0 0 6 0	6 6 0 6	6 6 8 6	1 1 7 9	1 1 7 1
9 9 4 9	0 0 5 0	5 5 3 5	8 8 9 8	7 9 7 9	7 7 1 7	1 1 6 1
2 7 2 7	8 8 5 8	2 2 7 8	6 6 1 6	6 5 6 5	4 4 9 4	6 0 6 0

The top printed digit is the right answer. The bottom two printed digits are the network's best two guesses.

The right answer is **almost** always in the top 2 guesses.

With model averaging they can now get about 25 errors.

How to detect a significant drop in the error rate

- Is 30 errors in 10,000 test cases significantly better than 40 errors?
 - It all depends on the particular errors!
 - The McNemar test uses the particular errors and can be much more powerful than a test that just uses the number of errors.

	model 1 wrong	model 1 right
model 2 wrong	29	1
model 2 right	11	9959

	model 1 wrong	model 1 right
model 2 wrong	15	15
model 2 right	25	9945

Neural Networks for Machine Learning

Lecture 5d

Convolutional neural networks for object recognition

Geoffrey Hinton

with

Nitish Srivastava

Kevin Swersky

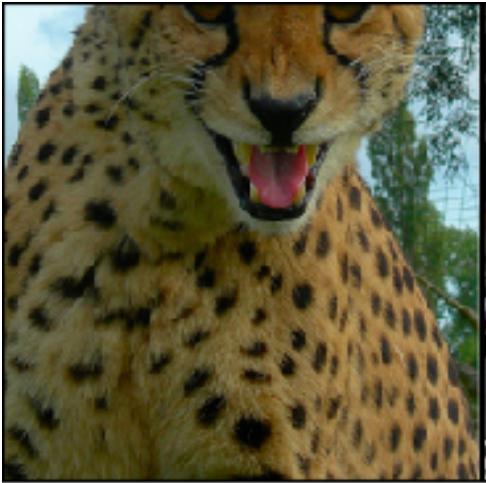
From hand-written digits to 3-D objects

- Recognizing real objects in color photographs downloaded from the web is much more complicated than recognizing hand-written digits:
 - Hundred times as many classes (1000 vs 10)
 - Hundred times as many pixels (256×256 color vs 28×28 gray)
 - Two dimensional image of three-dimensional scene.
 - Cluttered scenes requiring segmentation
 - Multiple objects in each image.
- Will the same type of convolutional neural network work?

The ILSVRC-2012 competition on ImageNet

- The dataset has 1.2 million high-resolution training images.
- The classification task:
 - Get the “correct” class in your top 5 bets. There are 1000 classes.
- The localization task:
 - For each bet, put a box around the object. Your box must have at least 50% overlap with the correct box.
- Some of the best existing computer vision methods were tried on this dataset by leading computer vision groups from Oxford, INRIA, XRCE, ...
 - Computer vision systems use complicated multi-stage systems.
 - The early stages are typically hand-tuned by optimizing a few parameters.

Examples from the test set (with the network's guesses)



cheetah

cheetah

leopard

snow leopard

Egyptian cat



bullet train

bullet train

passenger car

subway train

electric locomotive



hand glass

scissors

hand glass

frying pan

stethoscope

- University of Toronto (Alex Krizhevsky) • 16.4% 34.1%

Error rates on the ILSVRC-2012 competition

	classification	classification &localization
• University of Tokyo	• 26.1%	53.6%
• Oxford University Computer Vision Group	• 26.9%	50.0%
• INRIA (French national research institute in CS) + XRCE (Xerox Research Center Europe)	• 27.0%	
• University of Amsterdam	• 29.5%	

A neural network for ImageNet

- Alex Krizhevsky (NIPS 2012) developed a very deep convolutional neural net of the type pioneered by Yann Le Cun. Its architecture was:
 - 7 hidden layers not counting some max pooling layers.
 - The early layers were convolutional.
 - The last two layers were globally connected.
- The activation functions were:
 - Rectified linear units in every hidden layer. These train much faster and are more expressive than logistic units.
 - Competitive normalization to suppress hidden activities when nearby units have stronger activities. This helps with variations in intensity.

Tricks that significantly improve generalization

- Train on random 224x224 patches from the 256x256 images to get more data. Also use left-right reflections of the images.
 - At test time, combine the opinions from ten different patches: The four 224x224 corner patches plus the central 224x224 patch plus the reflections of those five patches.
- Use “dropout” to regularize the weights in the globally connected layers (which contain most of the parameters).
 - Dropout means that half of the hidden units in a layer are randomly removed for each training example.
 - This stops hidden units from relying too much on other hidden units.

The hardware required for Alex's net

- He uses a very efficient implementation of convolutional nets on two Nvidia GTX 580 Graphics Processor Units (over 1000 fast little cores)
 - GPUs are very good for matrix-matrix multiplies.
 - GPUs have very high bandwidth to memory.
 - This allows him to train the network in a week.
 - It also makes it quick to combine results from 10 patches at test time.
- We can spread a network over many cores if we can communicate the states fast enough.
- As cores get cheaper and datasets get bigger, big neural nets will improve faster than old-fashioned (*i.e.* pre Oct 2012) computer vision systems.

Finding roads in high-resolution images

- Vlad Mnih (ICML 2012) used a non-convolutional net with local fields and multiple layers of rectified linear units to find roads in cluttered aerial images.
 - It takes a large image patch and predicts a binary road label for the central 16x16 pixels.
 - There is lots of labeled training data available for this task.
- The task is hard for many reasons:
 - Occlusion by buildings trees and cars.
 - Shadows, Lighting changes
 - Minor viewpoint changes
- The worst problems are incorrect labels:
 - Badly registered maps
 - Arbitrary decisions about what counts as a road.
- Big neural nets trained on big image patches with millions of examples are the only hope.



The best road-finder
on the planet?



Neural Networks for Machine Learning

Lecture 6a

Overview of mini-batch gradient descent

Geoffrey Hinton

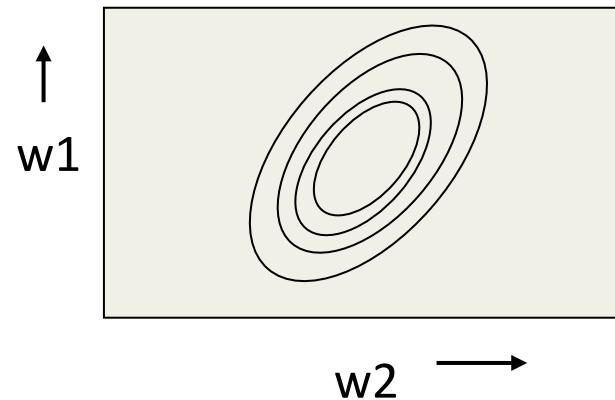
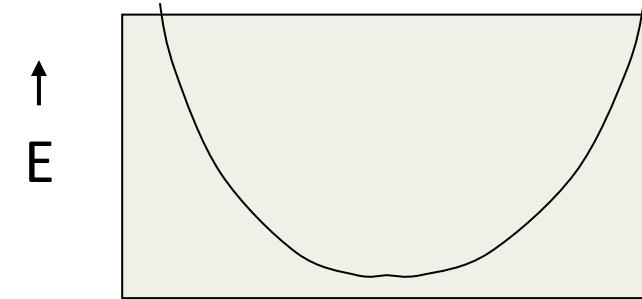
with

Nitish Srivastava

Kevin Swersky

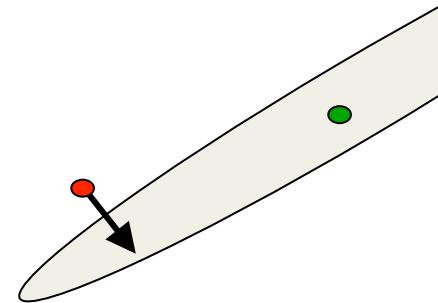
Reminder: The error surface for a linear neuron

- The error surface lies in a space with a horizontal axis for each weight and one vertical axis for the error.
 - For a linear neuron with a squared error, it is a quadratic bowl.
 - Vertical cross-sections are parabolas.
 - Horizontal cross-sections are ellipses.
- For multi-layer, non-linear nets the error surface is much more complicated.
 - But locally, a piece of a quadratic bowl is usually a very good approximation.



Convergence speed of full batch learning when the error surface is a quadratic bowl

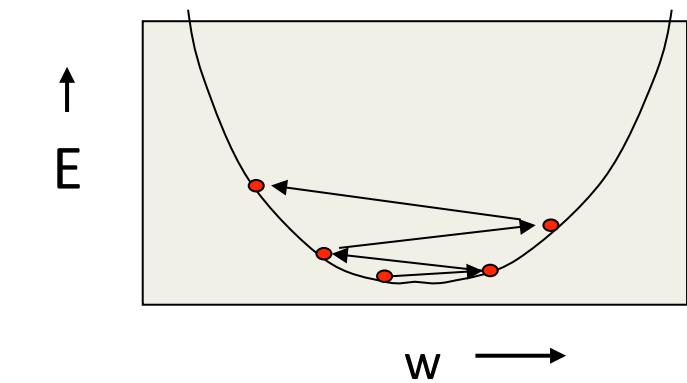
- Going downhill reduces the error, but the direction of steepest descent does not point at the minimum unless the ellipse is a circle.
 - The gradient is big in the direction in which we only want to travel a small distance.
 - The gradient is small in the direction in which we want to travel a large distance.



Even for non-linear multi-layer nets, the error surface is locally quadratic, so the same speed issues apply.

How the learning goes wrong

- If the learning rate is big, the weights slosh to and fro across the ravine.
 - If the learning rate is too big, this oscillation diverges.
- What we would like to achieve:
 - Move quickly in directions with small but consistent gradients.
 - Move slowly in directions with big but inconsistent gradients.



Stochastic gradient descent

- If the dataset is highly redundant, the gradient on the first half is almost identical to the gradient on the second half.
 - So instead of computing the full gradient, update the weights using the gradient on the first half and then get a gradient for the new weights on the second half.
 - The extreme version of this approach updates weights after each case. Its called “online”.
- Mini-batches are usually better than online.
 - Less computation is used updating the weights.
 - Computing the gradient for many cases simultaneously uses matrix-matrix multiplies which are very efficient, especially on GPUs
- Mini-batches need to be balanced for classes

Two types of learning algorithm

If we use the full gradient computed from all the training cases, there are many clever ways to speed up learning (e.g. non-linear conjugate gradient).

- The optimization community has studied the general problem of optimizing smooth non-linear functions for many years.
- Multilayer neural nets are not typical of the problems they study so their methods may need a lot of adaptation.

For large neural networks with very large and highly redundant training sets, it is nearly always best to use mini-batch learning.

- The mini-batches may need to be quite big when adapting fancy methods.
- Big mini-batches are more computationally efficient.

A basic mini-batch gradient descent algorithm

- Guess an initial learning rate.
 - If the error keeps getting worse or oscillates wildly, reduce the learning rate.
 - If the error is falling fairly consistently but slowly, increase the learning rate.
- Write a simple program to automate this way of adjusting the learning rate.
- Towards the end of mini-batch learning it nearly always helps to turn down the learning rate.
 - This removes fluctuations in the final weights caused by the variations between mini-batches.
- Turn down the learning rate when the error stops decreasing.
 - Use the error on a separate validation set

Neural Networks for Machine Learning

Lecture 6b

A bag of tricks for mini-batch gradient descent

Geoffrey Hinton

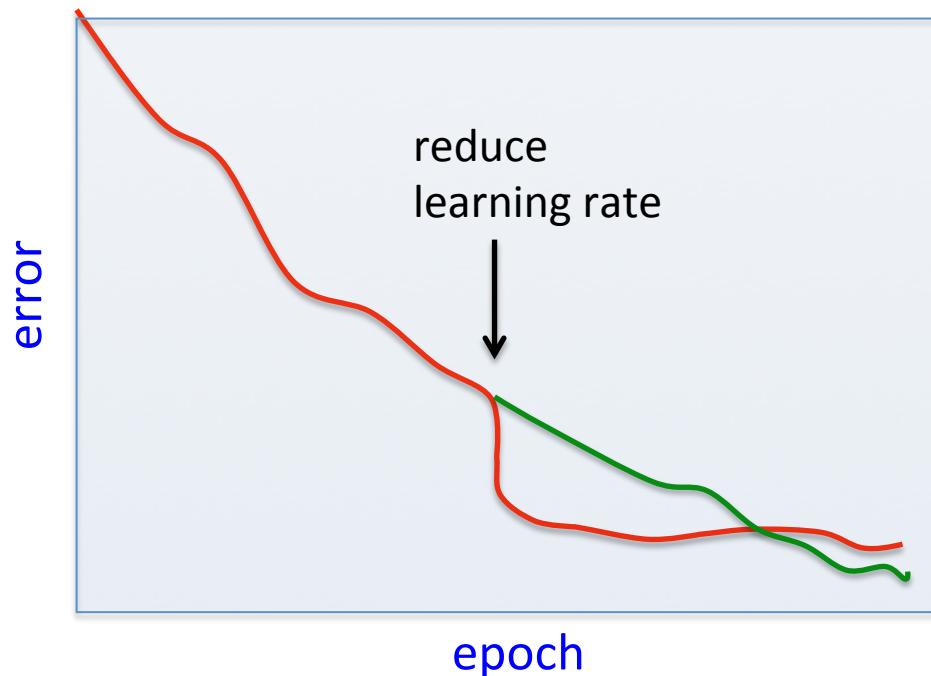
with

Nitish Srivastava

Kevin Swersky

Be careful about turning down the learning rate

- Turning down the learning rate reduces the random fluctuations in the error due to the different gradients on different mini-batches.
 - So we get a quick win.
 - But then we get slower learning.
- Don't turn down the learning rate too soon!

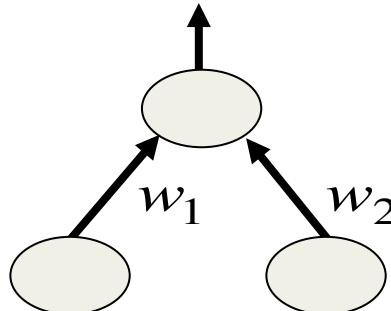


Initializing the weights

- If two hidden units have exactly the same bias and exactly the same incoming and outgoing weights, they will always get exactly the same gradient.
 - So they can never learn to be different features.
 - We break symmetry by initializing the weights to have small random values.
- If a hidden unit has a big fan-in, small changes on many of its incoming weights can cause the learning to overshoot.
 - We generally want smaller incoming weights when the fan-in is big, so initialize the weights to be proportional to $\text{sqrt}(\text{fan-in})$.
- We can also scale the learning rate the same way.

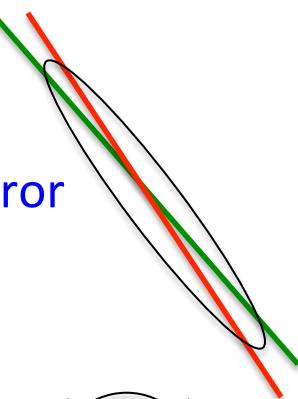
Shifting the inputs

- When using steepest descent, shifting the input values makes a big difference.
 - It usually helps to transform each component of the input vector so that it has zero mean over the whole training set.
- The hyperbolic tangent (which is $2 * \text{logistic} - 1$) produces hidden activations that are roughly zero mean.
 - In this respect it's better than the logistic.

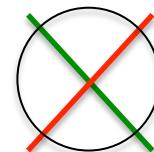


color indicates
training case

101, 101 → 2 gives error
101, 99 → 0 surface

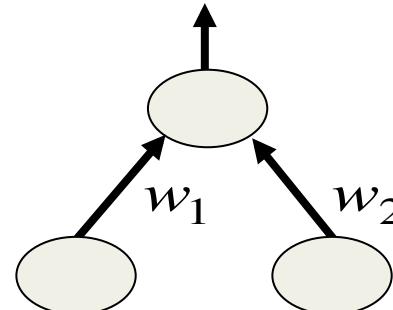


1, 1 → 2 gives error
1, -1 → 0 surface



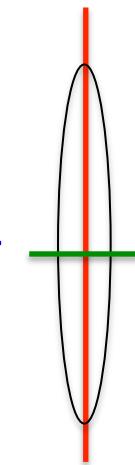
Scaling the inputs

- When using steepest descent, scaling the input values makes a big difference.
 - It usually helps to transform each component of the input vector so that it has unit variance over the whole training set.

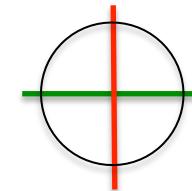


color indicates weight axis

$$\begin{array}{ll} 0.1, 10 \rightarrow 2 & \text{gives error} \\ 0.1, -10 \rightarrow 0 & \text{surface} \end{array}$$



$$\begin{array}{ll} 1, 1 \rightarrow 2 & \text{gives error} \\ 1, -1 \rightarrow 0 & \text{surface} \end{array}$$



A more thorough method: Decorrelate the input components

- For a linear neuron, we get a big win by decorrelating each component of the input from the other input components.
- There are several different ways to decorrelate inputs. A reasonable method is to use Principal Components Analysis.
 - Drop the principal components with the smallest eigenvalues.
 - This achieves some dimensionality reduction.
 - Divide the remaining principal components by the square roots of their eigenvalues. For a linear neuron, this converts an axis aligned elliptical error surface into a circular one.
- For a circular error surface, the gradient points straight towards the minimum.

Common problems that occur in multilayer networks

- If we start with a very big learning rate, the weights of each hidden unit will all become very big and positive or very big and negative.
 - The error derivatives for the hidden units will all become tiny and the error will not decrease.
 - This is usually a plateau, but people often mistake it for a local minimum.
- In classification networks that use a squared error or a cross-entropy error, the best guessing strategy is to make each output unit always produce an output equal to the proportion of time it should be a 1.
 - The network finds this strategy quickly and may take a long time to improve on it by making use of the input.
 - This is another plateau that looks like a local minimum.

Four ways to speed up mini-batch learning

- Use “momentum”
 - Instead of using the gradient to change the **position** of the weight “particle”, use it to change the **velocity**.
- Use separate adaptive learning rates for each parameter
 - Slowly adjust the rate using the consistency of the gradient for that parameter.
- **rmsprop:** Divide the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight.
 - This is the mini-batch version of just using the sign of the gradient.
- Take a fancy method from the optimization literature that makes use of curvature information (**not this lecture**)
 - Adapt it to work for neural nets
 - Adapt it to work for mini-batches.

Neural Networks for Machine Learning

Lecture 6c The momentum method

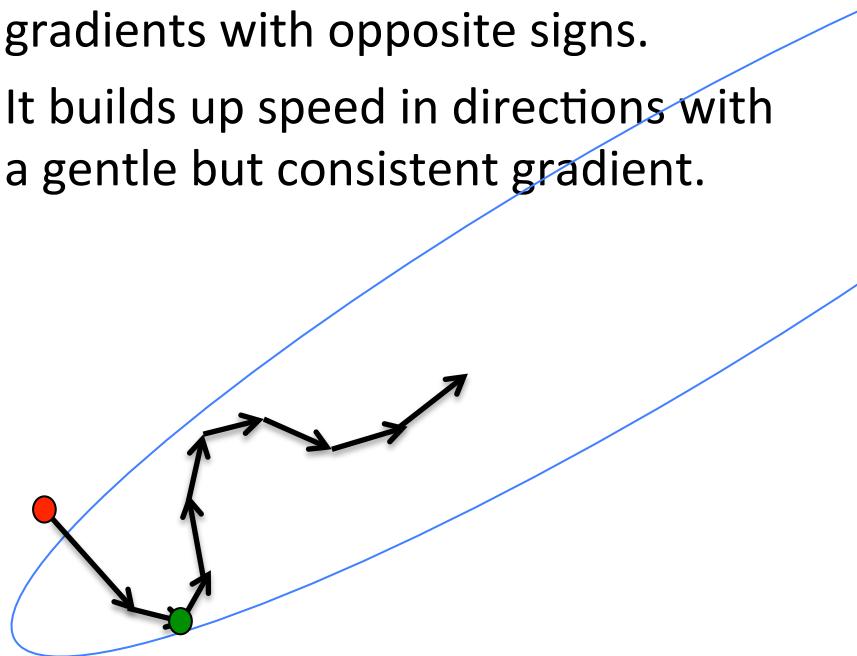
Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

The intuition behind the momentum method

Imagine a ball on the error surface. The location of the ball in the horizontal plane represents the weight vector.

- The ball starts off by following the gradient, but once it has velocity, it no longer does steepest descent.
- Its momentum makes it keep going in the previous direction.

- It damps oscillations in directions of high curvature by combining gradients with opposite signs.
- It builds up speed in directions with a gentle but consistent gradient.



The equations of the momentum method

$$\mathbf{v}(t) = \alpha \mathbf{v}(t-1) - \varepsilon \frac{\partial E}{\partial \mathbf{w}}(t)$$



The effect of the gradient is to increment the previous velocity. The velocity also decays by α which is slightly less than 1.

$$\Delta \mathbf{w}(t) = \mathbf{v}(t)$$



The weight change is equal to the current velocity.

$$= \alpha \mathbf{v}(t-1) - \varepsilon \frac{\partial E}{\partial \mathbf{w}}(t)$$

$$= \alpha \Delta \mathbf{w}(t-1) - \varepsilon \frac{\partial E}{\partial \mathbf{w}}(t)$$



The weight change can be expressed in terms of the previous weight change and the current gradient.

The behavior of the momentum method

- If the error surface is a tilted plane, the ball reaches a terminal velocity.
 - If the momentum is close to 1, this is much faster than simple gradient descent.
- At the beginning of learning there may be very large gradients.
 - So it pays to use a small momentum (e.g. 0.5).
 - Once the large gradients have disappeared and the weights are stuck in a ravine the momentum can be smoothly raised to its final value (e.g. 0.9 or even 0.99)
- This allows us to learn at a rate that would cause divergent oscillations without the momentum.

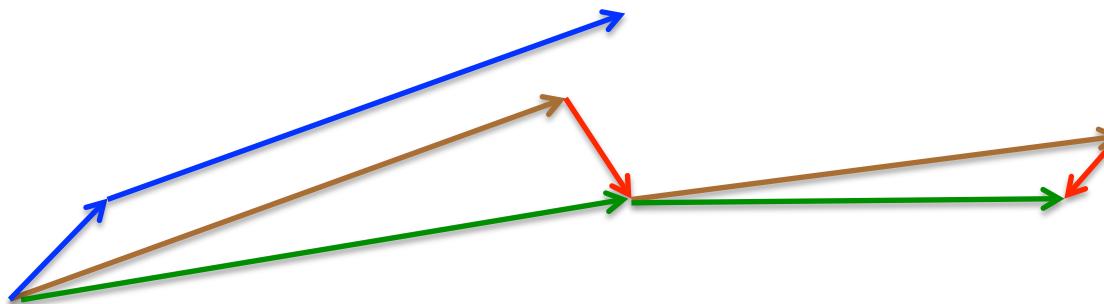
$$\mathbf{v}(\infty) = \frac{1}{1-\alpha} \left(-\varepsilon \frac{\partial E}{\partial \mathbf{w}} \right)$$

A better type of momentum (Nesterov 1983)

- The standard momentum method **first** computes the gradient at the current location and **then** takes a big jump in the direction of the updated accumulated gradient.
- Ilya Sutskever (2012 unpublished) suggested a new form of momentum that often works better.
 - Inspired by the Nesterov method for optimizing convex functions.
 - **First** make a big jump in the direction of the previous accumulated gradient.
 - **Then** measure the gradient where you end up and make a correction.
 - Its better to correct a mistake **after** you have made it!

A picture of the Nesterov method

- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.



brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

Neural Networks for Machine Learning

Lecture 6d

A separate, adaptive learning rate for each connection

Geoffrey Hinton

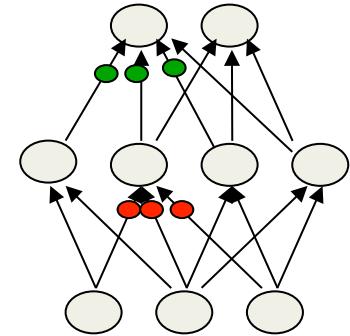
with

Nitish Srivastava

Kevin Swersky

The intuition behind separate adaptive learning rates

- In a multilayer net, the appropriate learning rates can vary widely between weights:
 - The magnitudes of the gradients are often very different for different layers, especially if the initial weights are small.
 - The fan-in of a unit determines the size of the “overshoot” effects caused by simultaneously changing many of the incoming weights of a unit to correct the same error.
- So use a global learning rate (set by hand) multiplied by an appropriate local gain that is determined empirically for each weight.



Gradients can get very small in the early layers of very deep nets.

The fan-in often varies widely between layers.

One way to determine the individual learning rates

- Start with a local gain of 1 for every weight.
- Increase the local gain if the gradient for that weight does not change sign.
- Use small additive increases and multiplicative decreases (for mini-batch)
 - This ensures that big gains decay rapidly when oscillations start.
 - If the gradient is totally random the gain will hover around 1 when we increase by plus δ half the time and decrease by times $1 - \delta$ half the time.

$$\Delta w_{ij} = -\varepsilon g_{ij} \frac{\partial E}{\partial w_{ij}}$$

if $\left(\frac{\partial E}{\partial w_{ij}}(t) \frac{\partial E}{\partial w_{ij}}(t-1) \right) > 0$

then $g_{ij}(t) = g_{ij}(t-1) + .05$

else $g_{ij}(t) = g_{ij}(t-1) * .95$

Tricks for making adaptive learning rates work better

- Limit the gains to lie in some reasonable range
 - e.g. [0.1, 10] or [.01, 100]
- Use full batch learning or big mini-batches
 - This ensures that changes in the sign of the gradient are not mainly due to the sampling error of a mini-batch.
- Adaptive learning rates can be combined with momentum.
 - Use the agreement in sign between the current gradient for a weight and the velocity for that weight (Jacobs, 1989).
- Adaptive learning rates only deal with axis-aligned effects.
 - Momentum does not care about the alignment of the axes.

Neural Networks for Machine Learning

Lecture 6e

rmsprop: Divide the gradient by a running average
of its recent magnitude

Geoffrey Hinton

with

Nitish Srivastava

Kevin Swersky

rprop: Using only the sign of the gradient

- The magnitude of the gradient can be very different for different weights and can change during learning.
 - This makes it hard to choose a single global learning rate.
- For **full batch learning**, we can deal with this variation by only using the sign of the gradient.
 - The weight updates are all of the same magnitude.
 - This escapes from plateaus with tiny gradients quickly.
- rprop: This combines the idea of only using the sign of the gradient with the idea of adapting the step size separately for each weight.
 - Increase the step size for a weight **multiplicatively** (e.g. times 1.2) if the signs of its last two gradients agree.
 - Otherwise decrease the step size multiplicatively (e.g. times 0.5).
 - Limit the step sizes to be less than 50 and more than a millionth (**Mike Shuster's advice**).

Why rprop does not work with mini-batches

- The idea behind stochastic gradient descent is that when the learning rate is small, it averages the gradients over successive mini-batches.
 - Consider a weight that gets a gradient of +0.1 on nine mini-batches and a gradient of -0.9 on the tenth mini-batch.
 - We want this weight to stay roughly where it is.
- rprop would increment the weight nine times and decrement it once by about the same amount (assuming any adaptation of the step sizes is small on this time-scale).
 - So the weight would grow a lot.
- Is there a way to combine:
 - The robustness of rprop.
 - The efficiency of mini-batches.
 - The effective averaging of gradients over mini-batches.

rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
 - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight
$$MeanSquare(w, t) = 0.9 \, MeanSquare(w, t-1) + 0.1 \left(\frac{\partial E}{\partial w}(t) \right)^2$$
- Dividing the gradient by $\sqrt{MeanSquare(w, t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

Further developments of rmsprop

- Combining rmsprop with standard momentum
 - Momentum does not help as much as it normally does. Needs more investigation.
- Combining rmsprop with Nesterov momentum (Sutskever 2012)
 - It works best if the RMS of the recent gradients is used to divide the correction rather than the jump in the direction of accumulated corrections.
- Combining rmsprop with adaptive learning rates for each connection
 - Needs more investigation.
- Other methods related to rmsprop
 - Yann LeCun's group has a fancy version in “No more pesky learning rates”

Summary of learning methods for neural networks

- For small datasets (e.g. 10,000 cases) or bigger datasets without much redundancy, use a full-batch method.
 - Conjugate gradient, LBFGS ...
 - adaptive learning rates, rprop ...
- For big, redundant datasets use mini-batches.
 - Try gradient descent with momentum.
 - Try rmsprop (with momentum ?)
 - Try LeCun's latest recipe.
- Why there is no simple recipe:
Neural nets differ a lot:
 - Very deep nets (especially ones with narrow bottlenecks).
 - Recurrent nets.
 - Wide shallow nets.**Tasks differ a lot:**
 - Some require very accurate weights, some don't.
 - Some have many very rare cases (e.g. words).

Neural Networks for Machine Learning

Lecture 7a

Modeling sequences: A brief overview

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

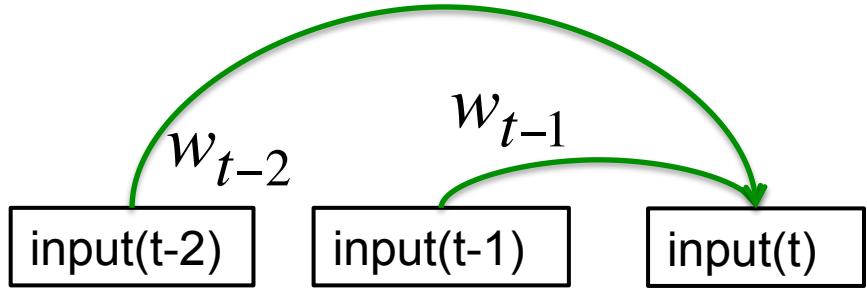
Getting targets when modeling sequences

- When applying machine learning to sequences, we often want to turn an input sequence into an output sequence that lives in a different domain.
 - *E. g. turn a sequence of sound pressures into a sequence of word identities.*
- When there is no separate target sequence, we can get a teaching signal by trying to predict the next term in the input sequence.
 - The target output sequence is the input sequence with an advance of 1 step.
 - This seems much more natural than trying to predict one pixel in an image from the other pixels, or one patch of an image from the rest of the image.
 - For temporal sequences there is a natural order for the predictions.
- Predicting the next term in a sequence blurs the distinction between supervised and unsupervised learning.
 - It uses methods designed for supervised learning, but it doesn't require a separate teaching signal.

Memoryless models for sequences

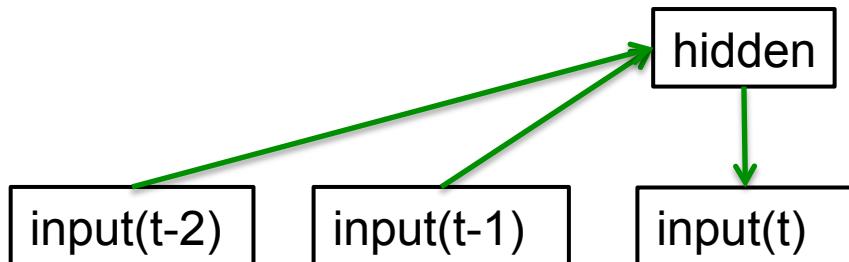
- **Autoregressive models**

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



- **Feed-forward neural nets**

These generalize autoregressive models by using one or more layers of non-linear hidden units.
e.g. Bengio's first language model.

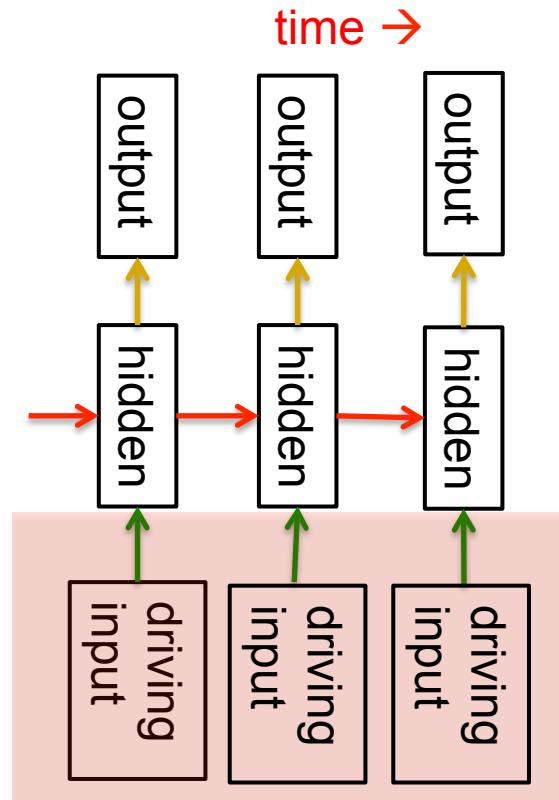


Beyond memoryless models

- If we give our generative model some hidden state, and if we give this hidden state its own internal dynamics, we get a much more interesting kind of model.
 - It can store information in its hidden state for a long time.
 - If the dynamics is noisy and the way it generates outputs from its hidden state is noisy, we can never know its exact hidden state.
 - The best we can do is to infer a probability distribution over the space of hidden state vectors.
- This inference is only tractable for two types of hidden state model.
 - The next three slides are mainly intended for people who already know about these two types of hidden state model. They show how RNNs differ.
 - Do not worry if you cannot follow the details.

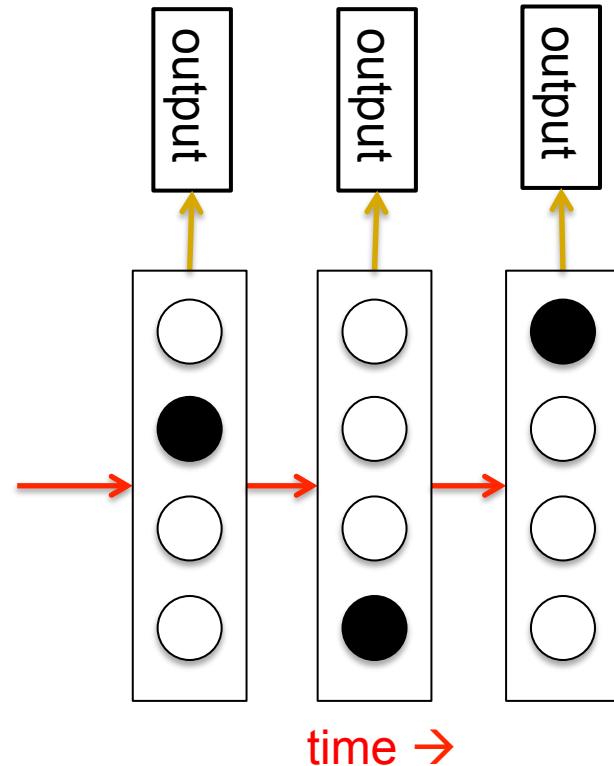
Linear Dynamical Systems (engineers love them!)

- These are generative models. They have a real-valued hidden state that cannot be observed directly.
 - The hidden state has linear dynamics with Gaussian noise and produces the observations using a linear model with Gaussian noise.
 - There may also be driving inputs.
- To predict the next output (so that we can shoot down the missile) we need to infer the hidden state.
 - A linearly transformed Gaussian is a Gaussian. So the distribution over the hidden state given the data so far is Gaussian!



Hidden Markov Models (computer scientists love them!)

- Hidden Markov Models have a discrete one-of-N hidden state. Transitions between states are stochastic and controlled by a transition matrix. The outputs produced by a state are stochastic.
 - We cannot be sure which state produced a given output. So the state is “hidden”.
 - It is easy to represent a probability distribution across N states with N numbers.
- To predict the next output we need to infer the probability distribution over hidden states.
 - HMMs have efficient algorithms for inference and learning.

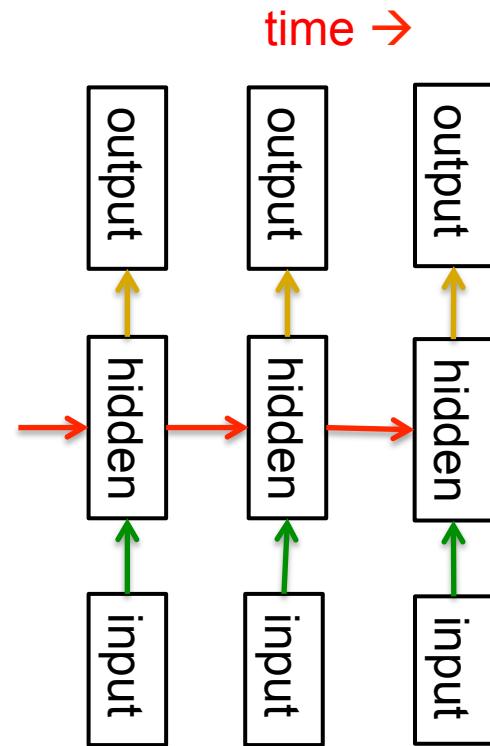


A fundamental limitation of HMMs

- Consider what happens when a hidden Markov model generates data.
 - At each time step it must select one of its hidden states. So with N hidden states it can only remember $\log(N)$ bits about what it generated so far.
- Consider the information that the first half of an utterance contains about the second half:
 - The syntax needs to fit (e.g. number and tense agreement).
 - The semantics needs to fit. The intonation needs to fit.
 - The accent, rate, volume, and vocal tract characteristics must all fit.
- All these aspects combined could be 100 bits of information that the first half of an utterance needs to convey to the second half. 2^{100}

Recurrent neural networks

- RNNs are very powerful, because they combine two properties:
 - Distributed hidden state that allows them to store a lot of information about the past efficiently.
 - Non-linear dynamics that allows them to update their hidden state in complicated ways.
- With enough neurons and time, RNNs can compute anything that can be computed by your computer.



Do generative models need to be stochastic?

- Linear dynamical systems and hidden Markov models are stochastic models.
 - But the posterior probability distribution over their hidden states given the observed data so far is a deterministic function of the data.
- Recurrent neural networks are deterministic.
 - So think of the hidden state of an RNN as the equivalent of the deterministic probability distribution over hidden states in a linear dynamical system or hidden Markov model.

Recurrent neural networks

- What kinds of behaviour can RNNs exhibit?
 - They can oscillate. Good for motor control?
 - They can settle to point attractors. Good for retrieving memories?
 - They can behave chaotically. Bad for information processing?
 - RNNs could potentially learn to implement lots of small programs that each capture a nugget of knowledge and run in parallel, interacting to produce very complicated effects.
- But the computational power of RNNs makes them very hard to train.
 - For many years we could not exploit the computational power of RNNs despite some heroic efforts (e.g. Tony Robinson's speech recognizer).

Neural Networks for Machine Learning

Lecture 7b Training RNNs with backpropagation

Geoffrey Hinton

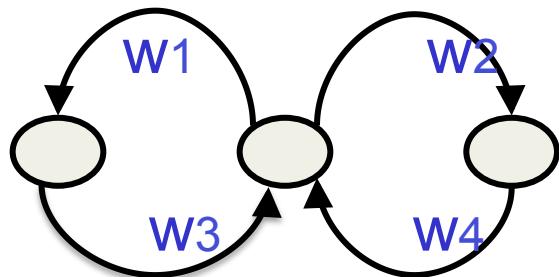
Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

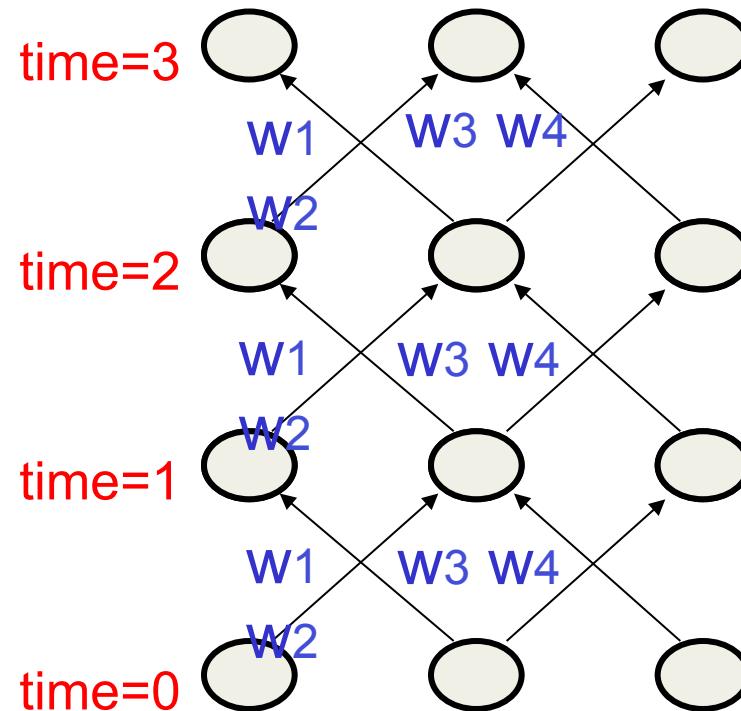
Abdel-rahman Mohamed

The equivalence between feedforward nets and recurrent nets



Assume that there is a time delay of 1 in using each connection.

The recurrent net is just a layered net that keeps reusing the same weights.



Reminder: Backpropagation with weight constraints

- It is easy to modify the backprop algorithm to incorporate linear constraints between the weights.
- We compute the gradients as usual, and then modify the gradients so that they satisfy the constraints.
 - So if the weights started off satisfying the constraints, they will continue to satisfy them.

To constrain: $w_1 = w_2$

we need: $\Delta w_1 = \Delta w_2$

compute: $\frac{\partial E}{\partial w_1}$ and $\frac{\partial E}{\partial w_2}$

use $\frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2}$ for w_1 and w_2

Backpropagation through time

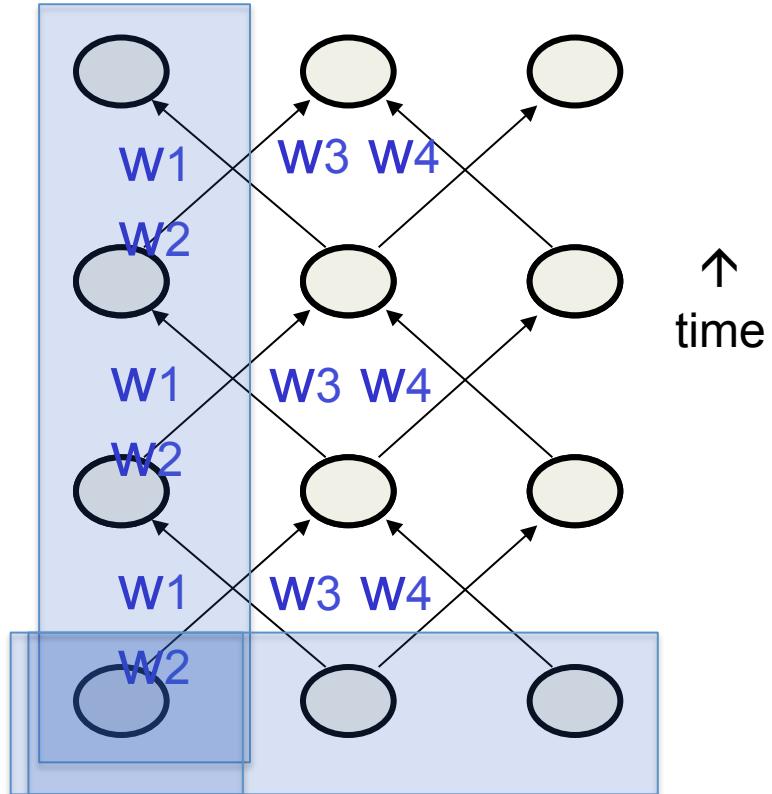
- We can think of the recurrent net as a layered, feed-forward net with shared weights and then train the feed-forward net with weight constraints.
- We can also think of this training algorithm in the time domain:
 - The forward pass builds up a stack of the activities of all the units at each time step.
 - The backward pass peels activities off the stack to compute the error derivatives at each time step.
 - After the backward pass we add together the derivatives at all the different times for each weight.

An irritating extra issue

- We need to specify the initial activity state of all the hidden and output units.
- We could just fix these initial states to have some default value like 0.5.
- But it is better to treat the initial states as learned parameters.
- We learn them in the same way as we learn the weights.
 - Start off with an initial random guess for the initial states.
 - At the end of each training sequence, backpropagate through time all the way to the initial states to get the gradient of the error function with respect to each initial state.
 - Adjust the initial states by following the negative gradient.

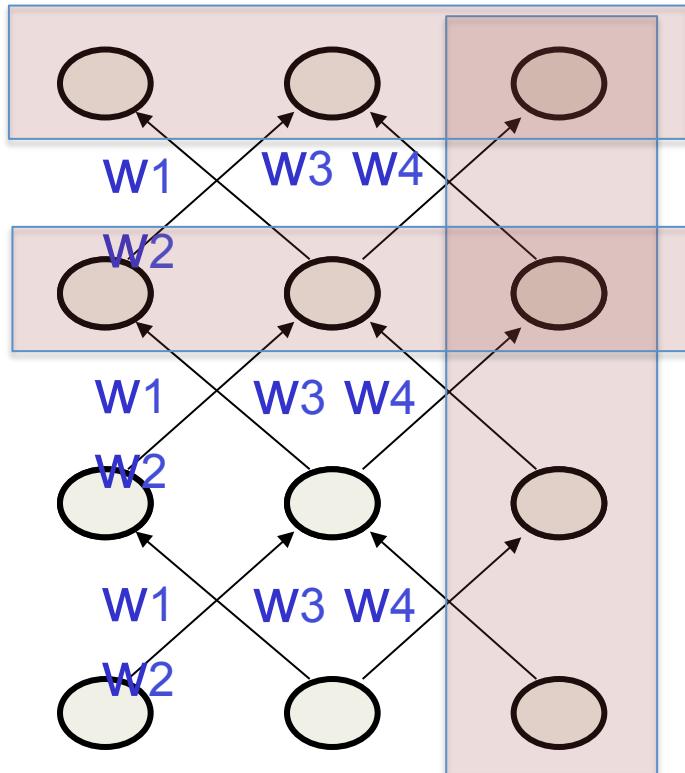
Providing input to recurrent networks

- We can specify inputs in several ways:
 - Specify the initial states of all the units.
 - Specify the initial states of a subset of the units.
 - Specify the states of the same subset of the units at every time step.
 - This is the natural way to model most sequential data.



Teaching signals for recurrent networks

- We can specify targets in several ways:
 - Specify desired final activities of all the units
 - Specify desired activities of all units for the last few steps
 - Good for learning attractors
 - It is easy to add in extra error derivatives as we backpropagate.
 - Specify the desired activity of a subset of the units.
 - The other units are input or hidden units.



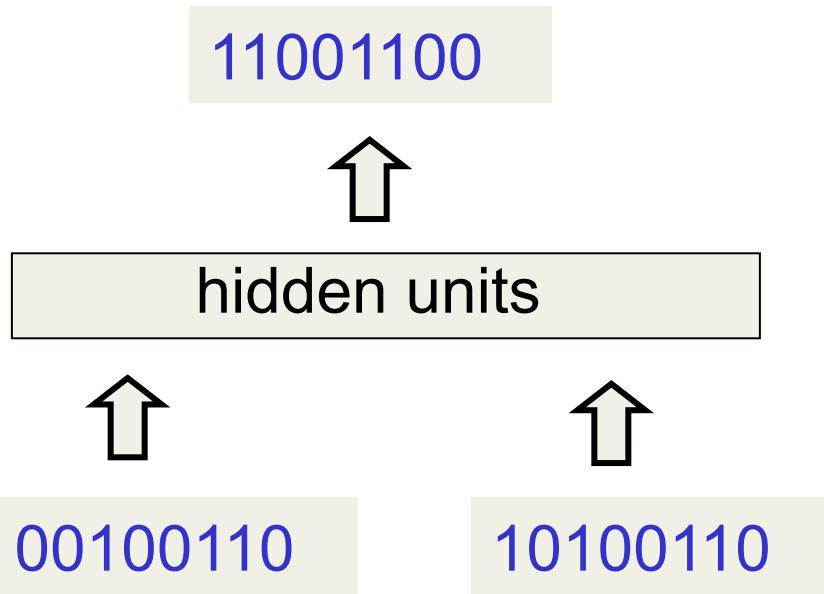
Neural Networks for Machine Learning

Lecture 7c A toy example of training an RNN

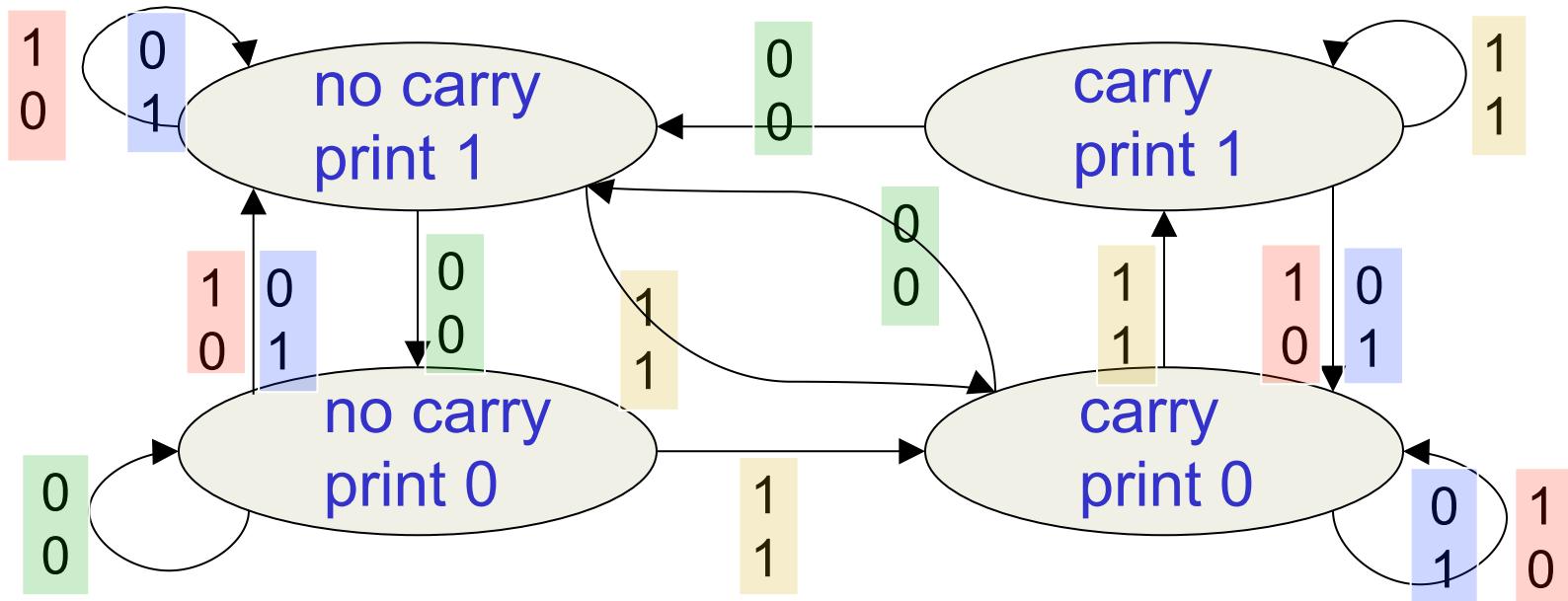
Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

A good toy problem for a recurrent network

- We can train a feedforward net to do binary addition, but there are obvious regularities that it cannot capture efficiently.
 - We must decide in advance the maximum number of digits in each number.
 - The processing applied to the beginning of a long number does not generalize to the end of the long number because it uses different weights.
- As a result, feedforward nets do not generalize well on the binary addition task.



The algorithm for binary addition



This is a finite state automaton. It decides what transition to make by looking at the next column. It prints after making the transition. It moves from right to left over the two input numbers.

A recurrent net for binary addition

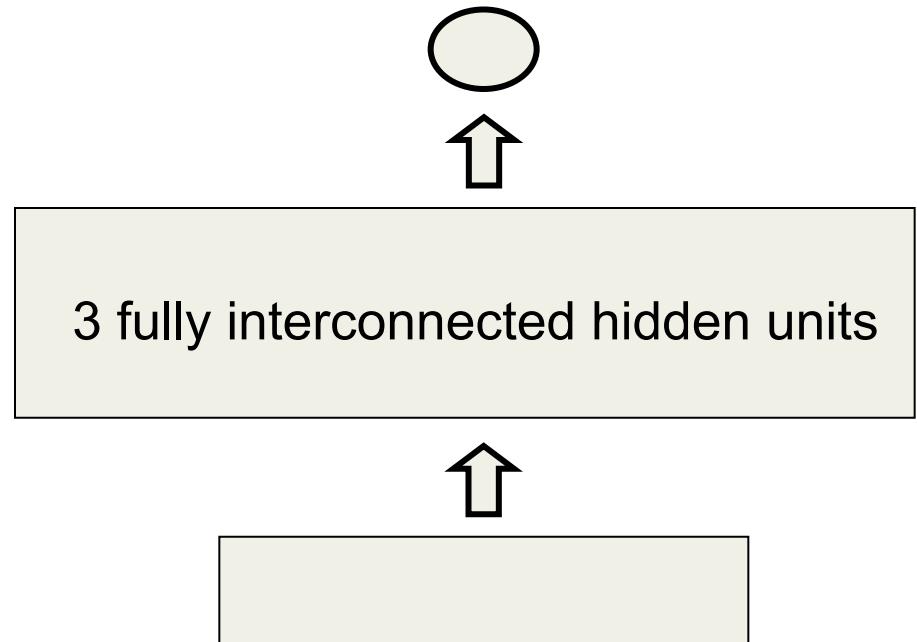
- The network has two input units and one output unit.
- It is given two input digits at each time step.
- The desired output at each time step is the output for the column that was provided as input two time steps ago.
 - It takes one time step to update the hidden units based on the two input digits.
 - It takes another time step for the hidden units to cause the output.

$$\begin{array}{r} 00110100 \\ 01001101 \\ \hline 10000001 \end{array}$$

← time

The connectivity of the network

- The 3 hidden units are fully interconnected in both directions.
 - This allows a hidden activity pattern at one time step to vote for the hidden activity pattern at the next time step.
- The input units have feedforward connections that allow them to vote for the next hidden activity pattern.



What the network learns

- It learns four distinct patterns of activity for the 3 hidden units. These **patterns** correspond to the nodes in the finite state automaton.
 - Do not confuse units in a neural network with nodes in a finite state automaton. Nodes are like activity vectors.
 - The automaton is restricted to be in exactly one **state** at each time. The hidden units are restricted to have exactly one **vector** of activity at each time.
- A recurrent network can emulate a finite state automaton, but it is exponentially more powerful. With N hidden neurons it has 2^N possible binary activity vectors (but only N^2 weights)
 - This is important when the input stream has two separate things going on at once.
 - A finite state automaton needs to square its number of states.
 - An RNN needs to double its number of **units**.

Neural Networks for Machine Learning

Lecture 7d

Why it is difficult to train an RNN

Geoffrey Hinton

Nitish Srivastava,

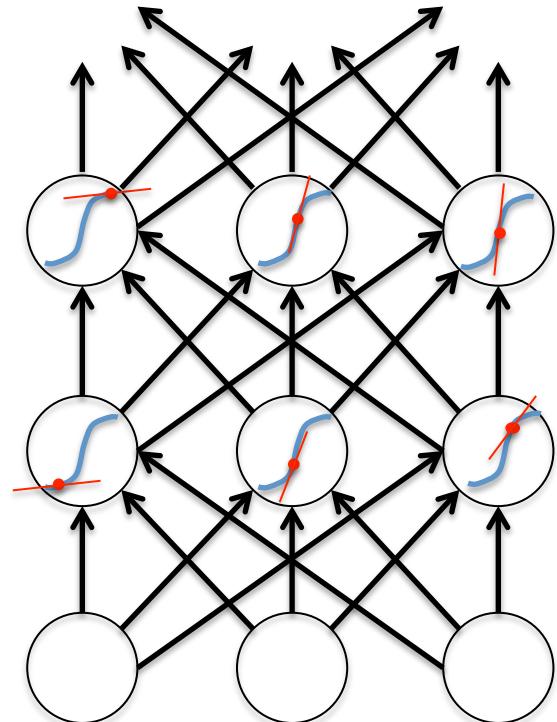
Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

The backward pass is linear

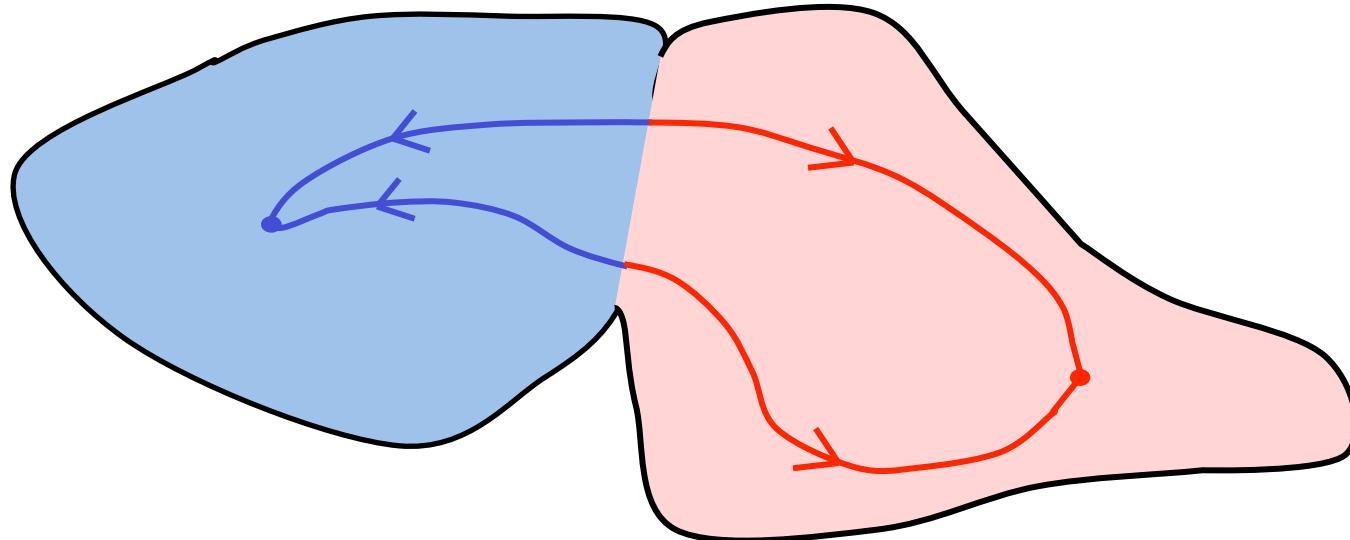
- There is a big difference between the forward and backward passes.
- In the forward pass we use squashing functions (like the logistic) to prevent the activity vectors from exploding.
- The backward pass, is completely **linear**. If you double the error derivatives at the final layer, all the error derivatives will double.
 - The forward pass determines the slope of the **linear** function used for backpropagating through each neuron.



The problem of exploding or vanishing gradients

- What happens to the magnitude of the gradients as we backpropagate through many layers?
 - If the weights are small, the gradients shrink exponentially.
 - If the weights are big the gradients grow exponentially.
- Typical feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.
- In an RNN trained on long sequences (e.g. 100 time steps) the gradients can easily explode or vanish.
 - We can avoid this by initializing the weights very carefully.
- Even with good initial weights, it's very hard to detect that the current target output depends on an input from many time-steps ago.
 - So RNNs have difficulty dealing with long-range dependencies.

Why the back-propagated gradient blows up



- If we start a trajectory within an attractor, small changes in where we start make no difference to where we end up.
- But if we start almost exactly on the boundary, tiny changes can make a huge difference.

Four effective ways to learn an RNN

- **Long Short Term Memory**
Make the RNN out of little modules that are designed to remember values for a long time.
- **Hessian Free Optimization:** Deal with the vanishing gradients problem by using a fancy optimizer that can detect directions with a tiny gradient but even smaller curvature.
 - The HF optimizer (Martens & Sutskever, 2011) is good at this.
- **Echo State Networks:** Initialize the input→hidden and hidden→hidden and output→hidden connections very carefully so that the hidden state has a huge reservoir of weakly coupled oscillators which can be selectively driven by the input.
 - ESNs only need to learn the hidden→output connections.
- **Good initialization with momentum**
Initialize like in Echo State Networks, but then learn all of the connections using momentum.

Neural Networks for Machine Learning

Lecture 7e Long term short term memory

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

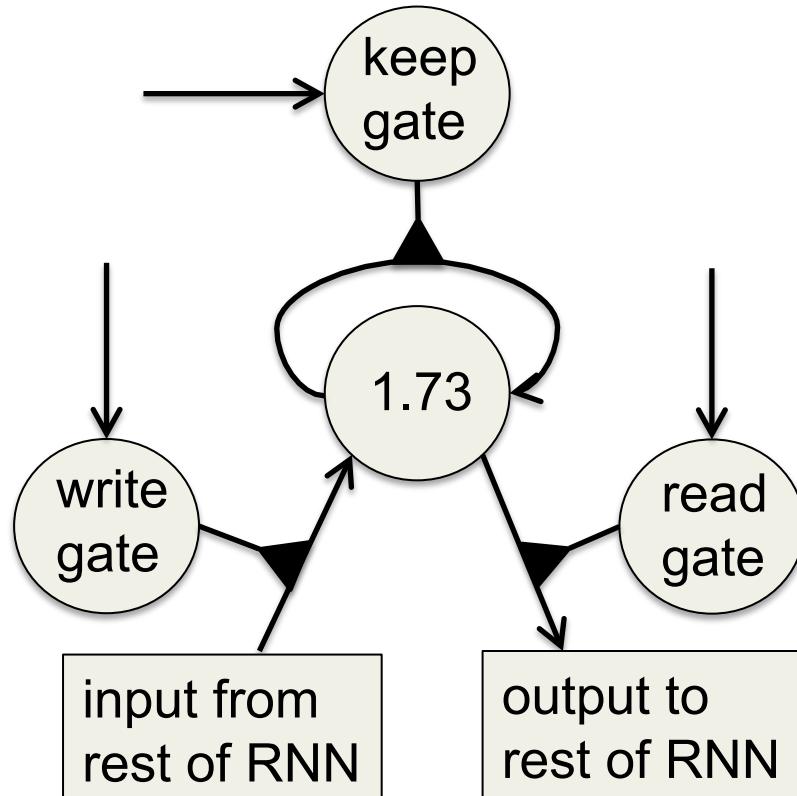
Long Short Term Memory (LSTM)

- Hochreiter & Schmidhuber (1997) solved the problem of getting an RNN to remember things for a long time (like hundreds of time steps).
- They designed a memory cell using logistic and linear units with multiplicative interactions.
- Information gets into the cell whenever its “write” gate is on.
- The information stays in the cell so long as its “keep” gate is on.
- Information can be read from the cell by turning on its “read” gate.

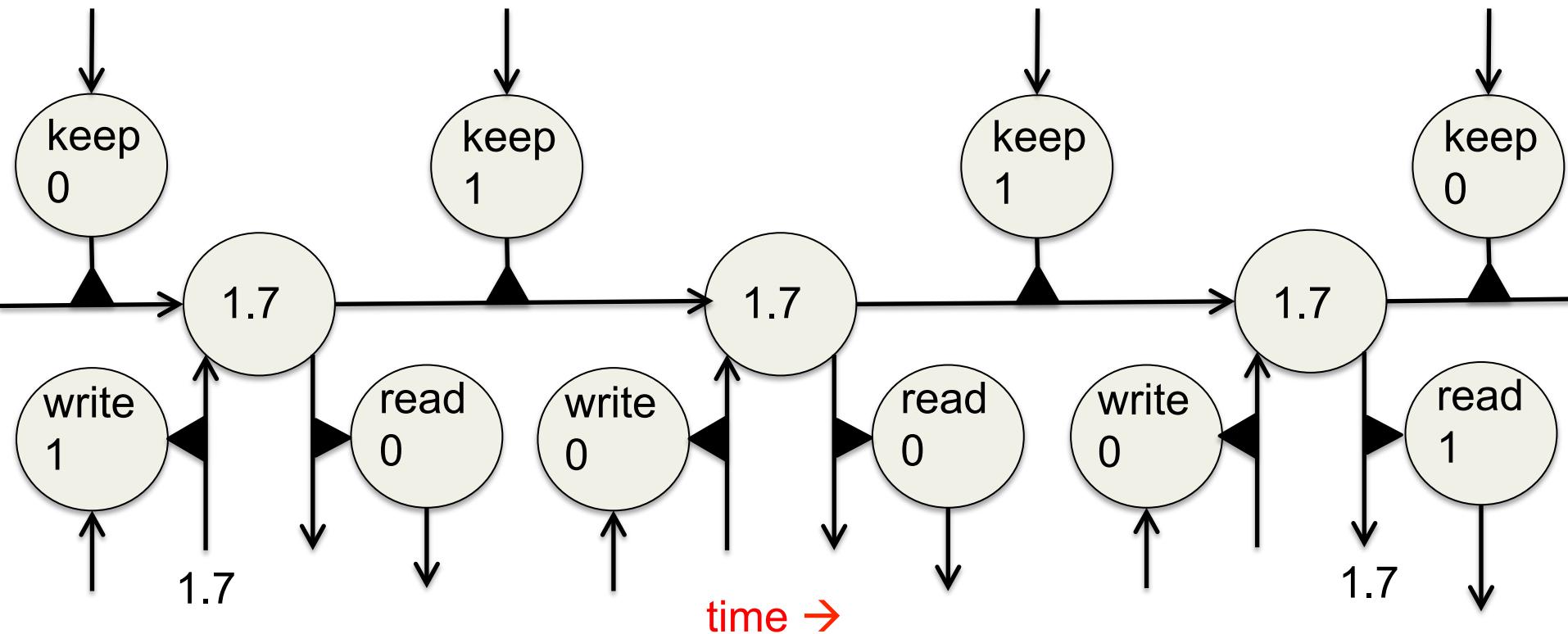
Implementing a memory cell in a neural network

To preserve information for a long time in the activities of an RNN, we use a circuit that implements an analog memory cell.

- A linear unit that has a self-link with a weight of 1 will maintain its state.
- Information is stored in the cell by activating its write gate.
- Information is retrieved by activating the read gate.
- We can backpropagate through this circuit because logistics are have nice derivatives.



Backpropagation through a memory cell



Reading cursive handwriting

- This is a natural task for an RNN.
- The input is a sequence of (x,y,p) coordinates of the tip of the pen, where p indicates whether the pen is up or down.
- The output is a sequence of characters.
- Graves & Schmidhuber (2009) showed that RNNs with LSTM are currently the best systems for reading cursive writing.
 - They used a sequence of small images as input rather than pen coordinates.

A demonstration of online handwriting recognition by an RNN with Long Short Term Memory (from Alex Graves)

- The movie that follows shows several different things:
- Row 1: This shows when the characters are recognized.
 - It never revises its output so difficult decisions are more delayed.
- Row 2: This shows the states of a subset of the memory cells.
 - Notice how they get reset when it recognizes a character.
- Row 3: This shows the writing. The net sees the x and y coordinates.
 - Optical input actually works a bit better than pen coordinates.
- Row 4: This shows the gradient backpropagated all the way to the x and y inputs from the currently most active character.
 - This lets you see which bits of the data are influencing the decision.

WARNING: OPTIONAL EXTRA MATERIAL

- The material in this video is considerably more difficult than in most of the other videos. I have included it for those who want to get some idea of how the HF optimizer works.
- You do not need to understand how HF works in order to understand the remaining videos in lecture 8.
- The questions in the weekly quiz and the final test will not be about the material in this video, so you can safely skip it if you want.

Neural Networks for Machine Learning

Lecture 8a

A brief overview of “Hessian-Free” optimization

Geoffrey Hinton

Nitish Srivastava,

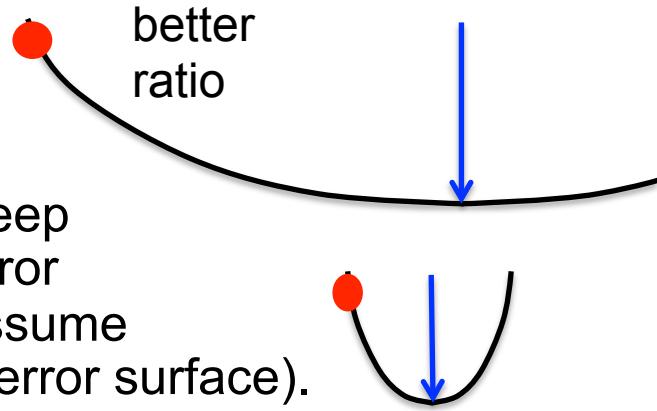
Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

How much can we reduce the error by moving in a given direction?

- If we choose a direction to move in and we keep going in that direction, how much does the error decrease before it starts rising again? We assume the curvature is constant (*i.e.* it's a quadratic error surface).
 - Assume the magnitude of the gradient decreases as we move down the gradient (*i.e.* the error surface is convex upward).
- The maximum error reduction depends on the ratio of the gradient to the curvature. So a good direction to move in is one with a high ratio of gradient to curvature, even if the gradient itself is small.
 - How can we find directions like these?



Newton's method

- The basic problem with steepest descent on a quadratic error surface is that the gradient is not the direction we want to go in.
 - If the error surface has circular cross-sections, the gradient is fine.
 - So let's apply a linear transformation that turns ellipses into circles.
- Newton's method multiplies the gradient vector by the inverse of the curvature matrix, H :

$$\Delta \mathbf{w} = -\varepsilon H(\mathbf{w})^{-1} \frac{dE}{d\mathbf{w}}$$

- On a real quadratic surface it jumps to the minimum in one step.
- Unfortunately, with only a million weights, the curvature matrix has a trillion terms and it is totally infeasible to invert it.

Curvature Matrices

- Each element in the curvature matrix specifies how the gradient in one direction changes as we move in some other direction.
 - The off-diagonal terms correspond to twists in the error surface.
- The reason steepest descent goes wrong is that the gradient for one weight gets messed up by the simultaneous changes to all the other weights.
 - The curvature matrix determines the sizes of these interactions.

	i	j	k
i	$\frac{\partial \left(\frac{\partial E}{\partial w_i} \right)}{\partial w_j}$		
j	$\frac{\partial \left(\frac{\partial E}{\partial w_j} \right)}{\partial w_i}$		
k			$\frac{\partial^2 E}{\partial w_k^2}$

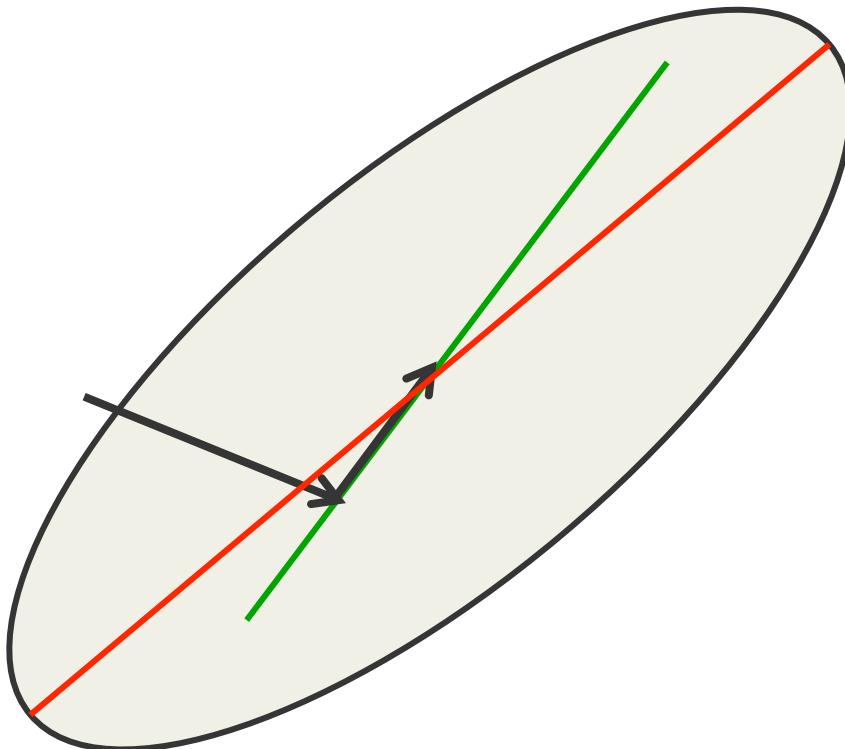
How to avoid inverting a huge matrix

- The curvature matrix has too many terms to be of use in a big network.
 - Maybe we can get some benefit from just using the terms along the leading diagonal (Le Cun). But the diagonal terms are only a tiny fraction of the interactions (they are the self-interactions).
- The curvature matrix can be approximated in many different ways
 - Hessian-free methods, LBFGS, ...
- In the HF method, we make an approximation to the curvature matrix and then, assuming that approximation is correct, we minimize the error using an efficient technique called conjugate gradient. Then we make another approximation to the curvature matrix and minimize again.
 - For RNNs its important to add a penalty for changing any of the hidden activities too much.

Conjugate gradient

- There is an alternative to going to the minimum in one step by multiplying by the inverse of the curvature matrix.
- Use a sequence of steps each of which finds the minimum along one direction.
- Make sure that each new direction is “conjugate” to the previous directions so you do not mess up the minimization you already did.
 - “conjugate” means that as you go in the new direction, you do not change the gradients in the previous directions.

A picture of conjugate gradient



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.

What does conjugate gradient achieve?

- After N steps, conjugate gradient is guaranteed to find the minimum of an N -dimensional **quadratic** surface. Why?
 - After many less than N steps it has typically got the error very close to the minimum value.
- Conjugate gradient can be applied directly to a non-quadratic error surface and it usually works quite well (**non-linear conjugate grad.**)
- The HF optimizer uses conjugate gradient for minimization on a genuinely quadratic surface where it excels.
 - The genuinely quadratic surface is the quadratic approximation to the true surface.

Neural Networks for Machine Learning

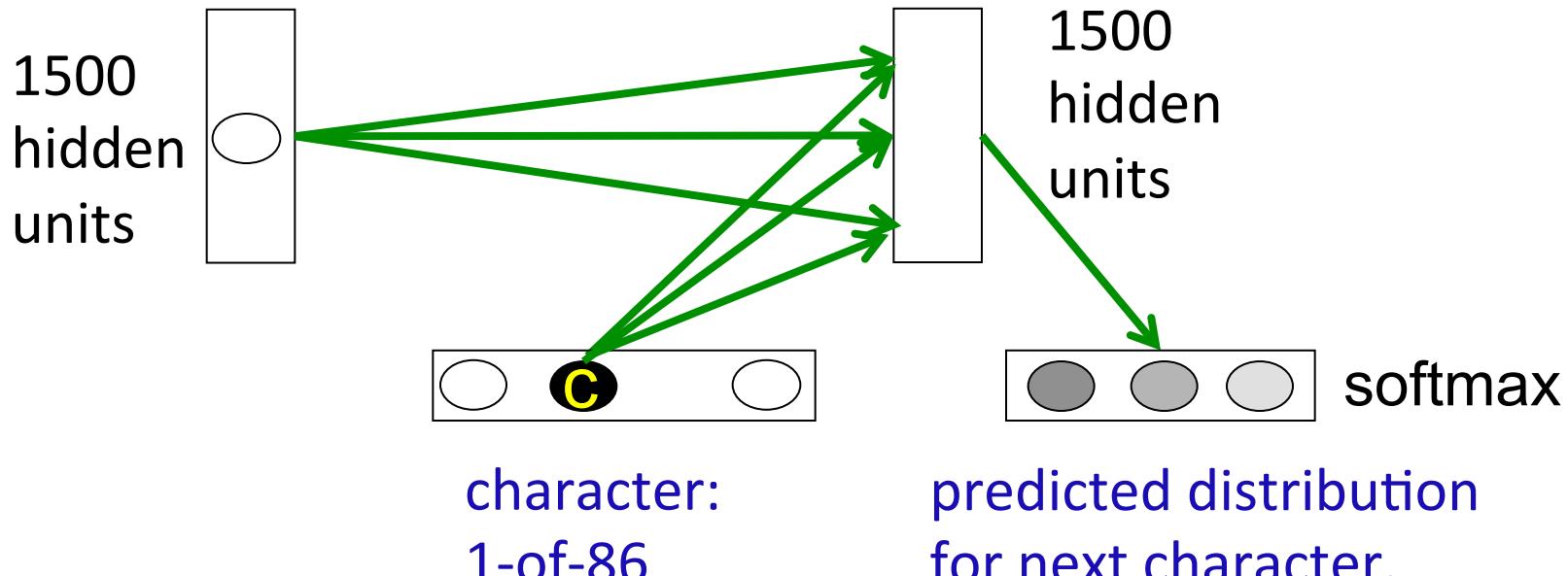
Lecture 8b Modeling character strings with multiplicative connections

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

Modeling text: Advantages of working with characters

- The web is composed of character strings.
- Any learning method powerful enough to understand the world by reading the web ought to find it trivial to learn which strings make words (this turns out to be true, as we shall see).
- Pre-processing text to get words is a big hassle
 - What about morphemes (prefixes, suffixes etc)
 - What about subtle effects like “sn” words?
 - What about New York?
 - What about Finnish
 - ymmärtämättömyydellänsäkaän

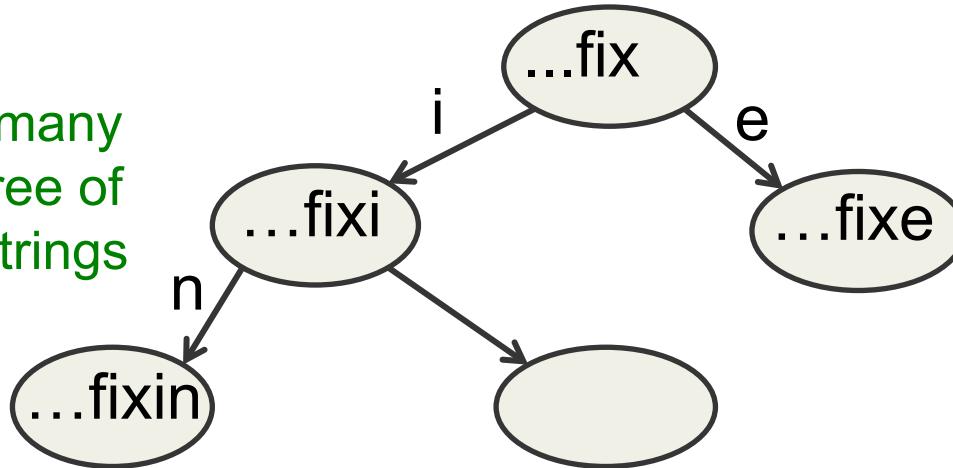
An obvious recurrent neural net



It's a lot easier to predict 86 characters than 100,000 words.

A sub-tree in the tree of all character strings

There are exponentially many nodes in the tree of all character strings of length N.



In an RNN, each node is a hidden state vector. The next character must transform this to a new node.

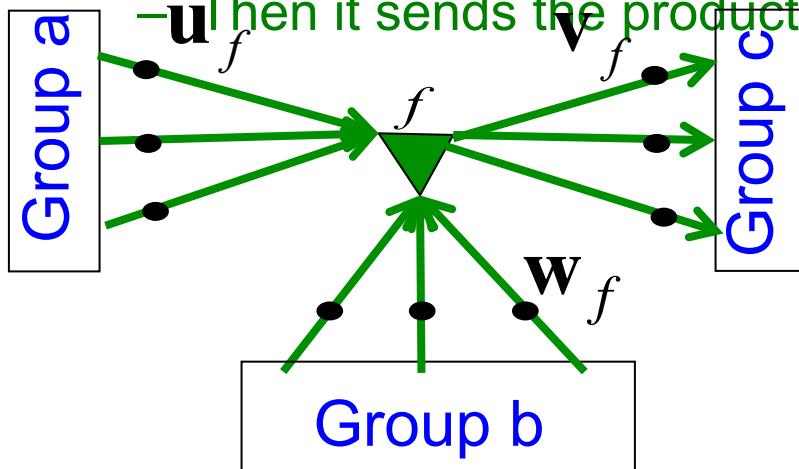
- If the nodes are implemented as hidden states in an RNN, different nodes can share structure because they use distributed representations.
- The next hidden representation needs to depend on the **conjunction** of the current character and the current hidden representation.

Multiplicative connections

- Instead of using the inputs to the recurrent net to provide additive extra input to the hidden units, we could use the current input character to choose the whole hidden-to-hidden weight matrix.
 - But this requires $86 \times 1500 \times 1500$ parameters
 - This could make the net overfit.
- Can we achieve the same kind of multiplicative interaction using fewer parameters?
 - We want a different transition matrix for each of the 86 characters, but we want these 86 character-specific weight matrices to share parameters (the characters 9 and 8 should have similar matrices).

Using factors to implement multiplicative interactions

- We can get groups a and b to interact multiplicatively by using “factors”.
 - Each factor first computes a weighted sum for each of its input groups.
 - Then it sends the product of the weighted sums to its output group.

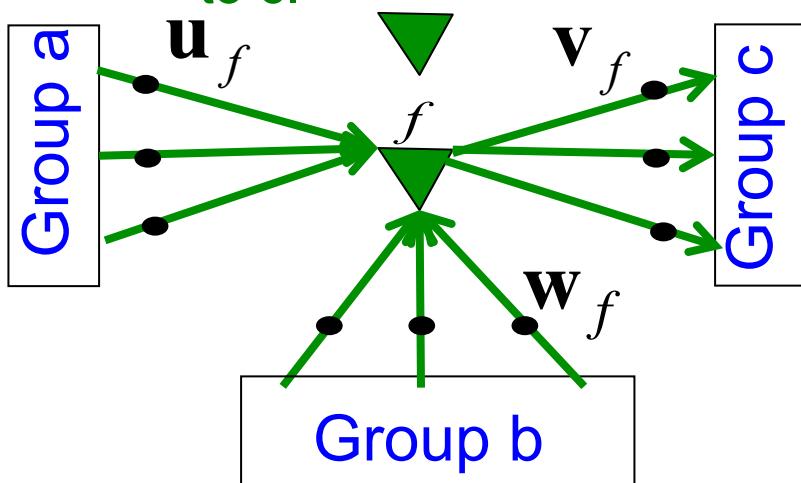


$$\mathbf{c}_f = (\mathbf{b}^T \mathbf{w}_f) (\mathbf{a}^T \mathbf{u}_f) \mathbf{v}_f$$

↑ ↑ ↑
vector of scalar scalar
inputs to input to f input to f
group c from group b from group a

Using factors to implement a set of basis matrices

- We can think about factors another way:
 - Each factor defines a rank 1 transition matrix from a to c.



$$c_f = (b^T w_f)(a^T u_f) v_f$$

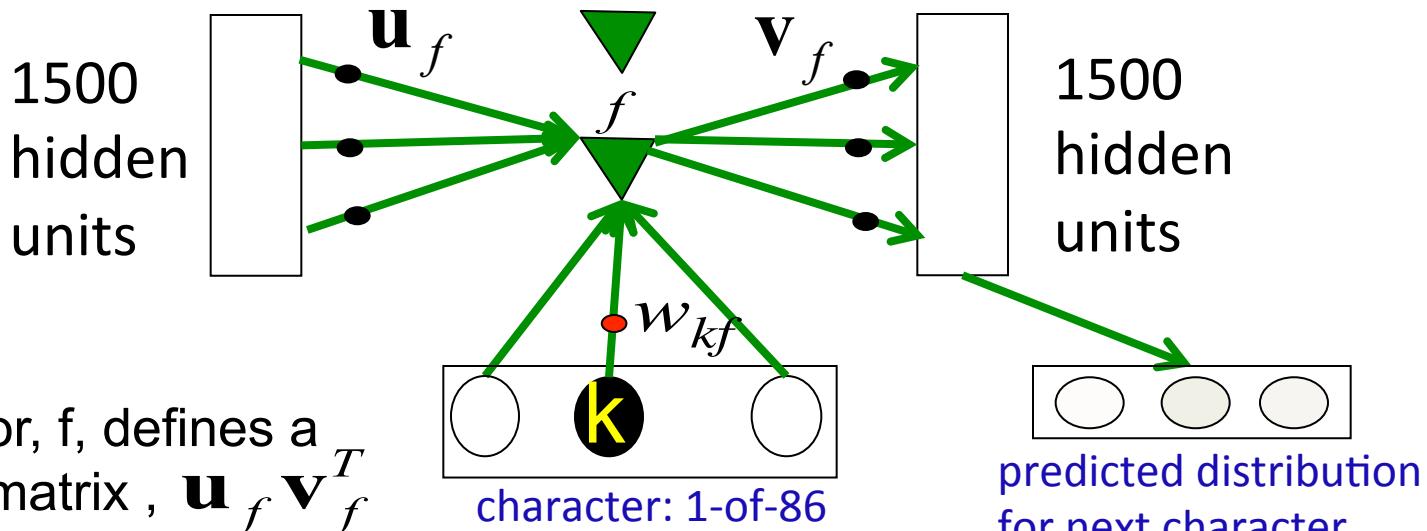
$$c_f = \boxed{(b^T w_f)} \quad \boxed{(u_f v_f^T)} \quad a$$

scalar
coefficient

outer product
transition
matrix with
rank 1

$$c = \left(\sum_f (b^T w_f)(u_f v_f^T) \right) a$$

Using 3-way factors to allow a character to create a whole transition matrix



Each character, k , determines a **gain** w_{kf} for each of these matrices.

Neural Networks for Machine Learning

Lecture 8c

Learning to predict the next character using HF

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

Training the character model

- Ilya Sutskever used 5 million strings of 100 characters taken from wikipedia. For each string he starts predicting at the 11th character.
- Using the HF optimizer, it took a month on a GPU board to get a really good model.
- Ilya's current best RNN is probably the best single model for character prediction (**combinations of many models do better**).
- It works in a very different way from the best other models.
 - It can balance quotes and brackets over long distances. Models that rely on matching previous contexts cannot do this.

How to generate character strings from the model

- Start the model with its default hidden state.
- Give it a “burn-in” sequence of characters and let it update its hidden state after each character.
- Then look at the probability distribution it predicts for the next character.
- Pick a character randomly from that distribution and tell the net that this was the character that actually occurred.
 - i.e. tell it that its guess was correct, whatever it guessed.
- Continue to let it pick characters until bored.
- Look at the character strings it produces to see what it “knows”.

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 **ephemeral** 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."

Some completions produced by the model

- Sheila thrunges (most frequent)
- People thrunge (most frequent next character is space)
- Shiela, Thrungelini del Rey (first try)
- The meaning of life is literary recognition. (6th try)

- 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?

- It knows a huge number of words and a lot about proper names, dates, and numbers.
- It is good at balancing quotes and brackets.
 - It can count brackets: none, one, many
- It knows a lot about syntax but its very hard to pin down exactly what form this knowledge has.
 - Its syntactic knowledge is not modular.
- It knows a lot of weak semantic associations
 - E.g. it knows Plato is associated with Wittgenstein and cabbage is associated with vegetable.

RNNs for predicting the next word

- Tomas Mikolov and his collaborators have recently trained quite large RNNs on quite large training sets using BPTT.
 - 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.
- RNNs require much less training data to reach the same level of performance as other models.
- RNNs improve faster than other methods as the dataset gets bigger.
 - This is going to make them very hard to beat.

Neural Networks for Machine Learning

Lecture 8d Echo state networks

Geoffrey Hinton

Nitish Srivastava,

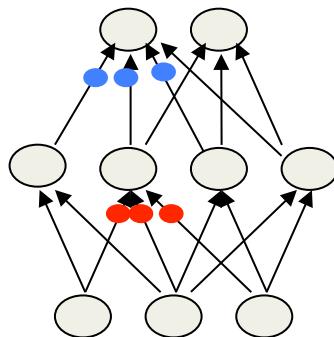
Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

The key idea of echo state networks (perceptrons again?)

- A very simple way to learn a feedforward network is to make the early layers random and fixed.
- Then 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.



- The equivalent idea for RNNs is to fix the **input→hidden** connections and the **hidden→hidden** connections at random values and only learn the **hidden→output** connections.
 - The learning is then very simple (assuming linear output units).
 - Its important to set the random connections very carefully so the RNN does not explode or die.

Setting the random connections in an Echo State Network

- Set the hidden \rightarrow hidden weights so that the length of the activity vector stays about the same after each iteration.
 - This allows the input to echo around the network for a long time.
- Use sparse connectivity (*i.e.* set most of the weights to zero).
 - This creates lots of loosely coupled oscillators.
- Choose the scale of the input \rightarrow hidden connections very carefully.
 - They need to drive the loosely coupled oscillators without wiping out the information from the past that they already contain.
- The learning is so fast that we can try many different scales for the weights and sparsenesses.
 - This is often necessary.

A simple example of an echo state network

INPUT SEQUENCE

A real-valued time-varying value that specifies the frequency of a sine wave.

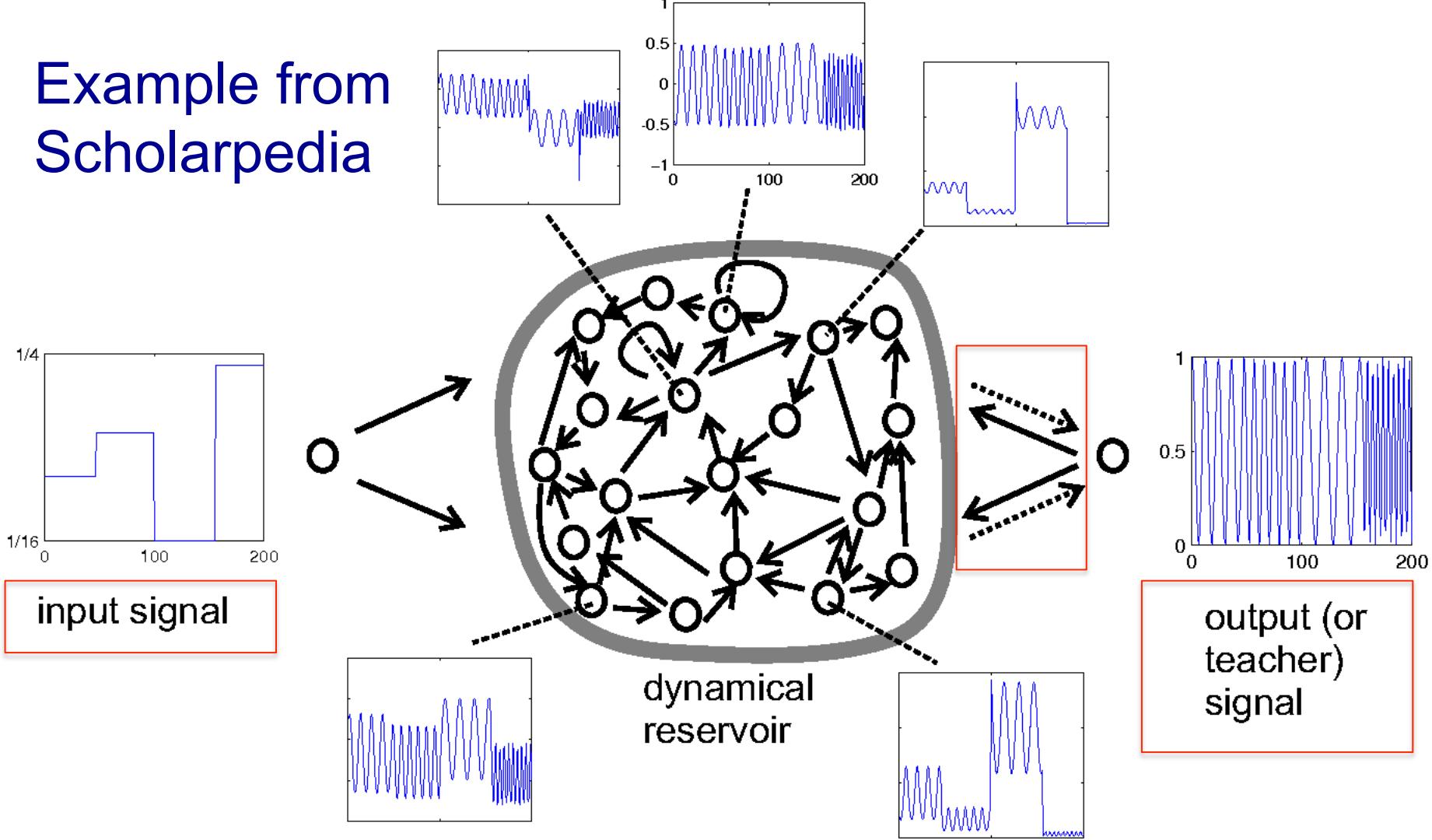
TARGET OUTPUT SEQUENCE

A sine wave with the currently specified frequency.

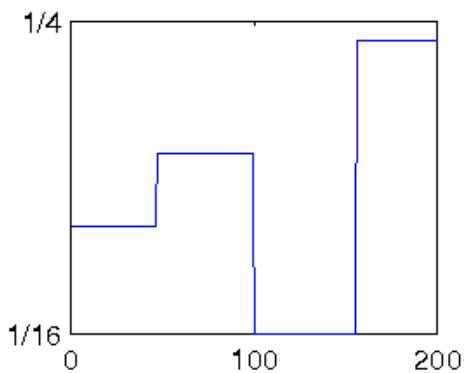
LEARNING METHOD

Fit a linear model that takes the states of the hidden units as input and produces a single scalar output.

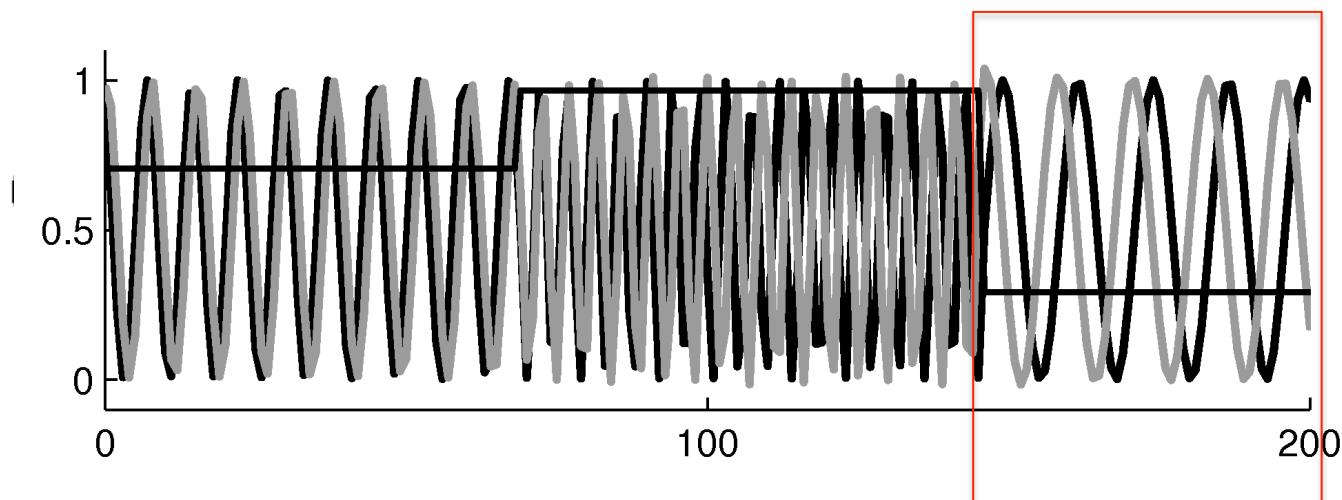
Example from Scholarpedia



The target and predicted outputs after learning



input signal



Beyond echo state networks

- **Good aspects of ESNs**

Echo state networks can be trained very fast because they just fit a linear model.

- They demonstrate that it's very important to initialize weights sensibly.

- They can do impressive modeling of one-dimensional time-series.

- but they cannot compete seriously for high-dimensional data like pre-processed speech.

- **Bad aspects of ESNs**

They need many more hidden units for a given task than an RNN that learns the hidden \rightarrow hidden weights.

- Ilya Sutskever (2012) has shown that if the weights are initialized using the ESN methods, RNNs can be trained very effectively.

- He uses rmsprop with momentum.

Neural Networks for Machine Learning

Lecture 9a

Overview of ways to improve generalization

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

Reminder: Overfitting

- The training data contains information about the regularities in the mapping from input to output. But it also contains sampling error.
 - There will be accidental regularities just because of the particular training cases that were chosen.
- When we fit the model, it cannot tell which regularities are real and which are caused by sampling error.
 - So it fits both kinds of regularity. If the model is very flexible it can model the sampling error really well.

Preventing overfitting

- Approach 1: Get more data!
 - Almost always the best bet if you have enough compute power to train on more data.
- Approach 2: Use a model that has the right capacity:
 - enough to fit the true regularities.
 - not enough to also fit spurious regularities (if they are weaker).
- Approach 3: Average many different models.
 - Use models with different forms.
 - Or train the model on different subsets of the training data (this is called “bagging”).
- Approach 4: (Bayesian) Use a single neural network architecture, but average the predictions made by many different weight vectors.

Some ways to limit the capacity of a neural net

- The capacity can be controlled in many ways:
 - Architecture: Limit the number of hidden layers and the number of units per layer.
 - Early stopping: Start with small weights and stop the learning before it overfits.
 - Weight-decay: Penalize large weights using penalties or constraints on their squared values (L2 penalty) or absolute values (L1 penalty).
 - Noise: Add noise to the weights or the activities.
- Typically, a combination of several of these methods is used.

How to choose meta parameters that control capacity (like the number of hidden units or the size of the weight penalty)

- The wrong method is to try lots of alternatives and see which gives the best performance on the test set.
 - This is easy to do, but it gives a false impression of how well the method works.
 - The settings that work best on the test set are unlikely to work as well on a new test set drawn from the same distribution.
- An extreme example:
Suppose the test set has random answers that do not depend on the input.
 - The best architecture will do better than chance on the test set.
 - But it cannot be expected to do better than chance on a new test set.

Cross-validation: A better way to choose meta parameters

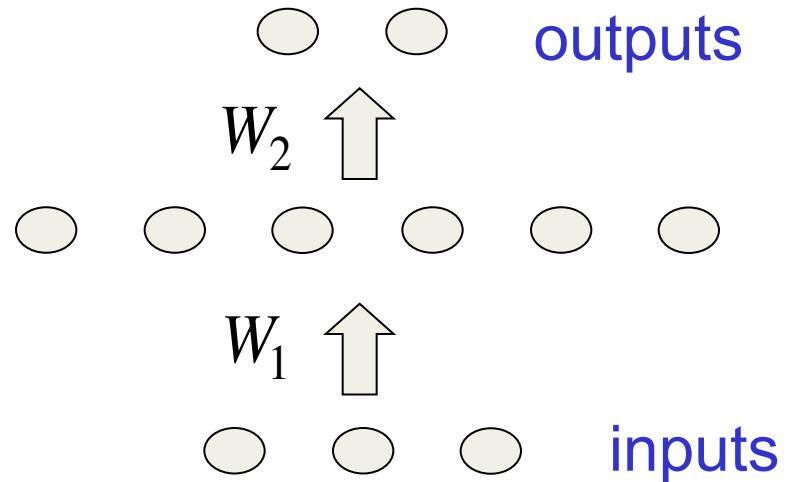
- Divide the total dataset into three subsets:
 - Training data is used for learning the parameters of the model.
 - Validation data is not used for learning but is used for deciding what settings of the meta parameters work best.
 - Test data is used to get a final, unbiased estimate of how well the network works. We expect this estimate to be worse than on the validation data.
- We could divide the total dataset into one final test set and N other subsets and train on all but one of those subsets to get N different estimates of the validation error rate.
 - This is called N -fold cross-validation.
 - The N estimates are not independent.

Preventing overfitting by early stopping

- If we have lots of data and a big model, it's very expensive to keep re-training it with different sized penalties on the weights.
- It is much cheaper to start with very small weights and let them grow until the performance on the validation set starts getting worse.
 - But it can be hard to decide when performance is getting worse.
- The capacity of the model is limited because the weights have not had time to grow big.

Why early stopping works

- When the weights are very small, every hidden unit is in its linear range.
 - So a net with a large layer of hidden units is linear.
 - It has no more capacity than a linear net in which the inputs are directly connected to the outputs!
- As the weights grow, the hidden units start using their non-linear ranges so the capacity grows.



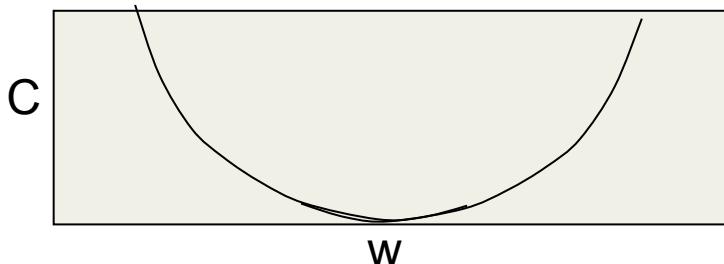
Neural Networks for Machine Learning

Lecture 9b Limiting the size of the weights

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

Limiting the size of the weights

- The standard L2 weight penalty involves adding an extra term to the cost function that penalizes the squared weights.
 - This keeps the weights small unless they have big error derivatives.



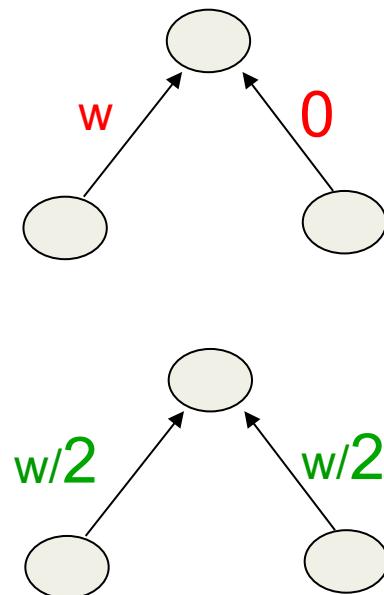
$$C = E + \frac{\lambda}{2} \sum_i w_i^2$$

$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$

when $\frac{\partial C}{\partial w_i} = 0, \quad w_i = -\frac{1}{\lambda} \frac{\partial E}{\partial w_i}$

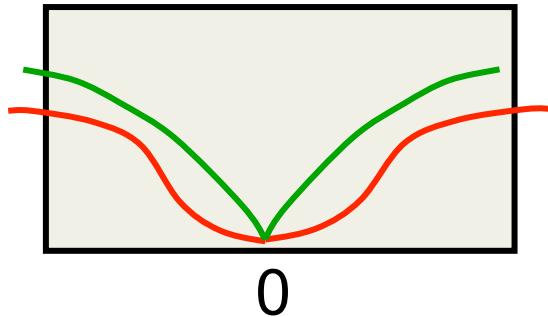
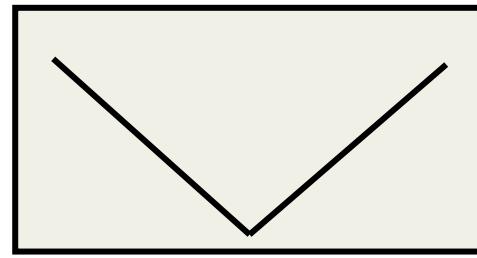
The effect of L2 weight cost

- It prevents the network from using weights that it does not need.
 - This can often improve generalization a lot because it helps to stop the network from fitting the sampling error.
 - It makes a smoother model in which the output changes more slowly as the input changes.
- If the network has two very similar inputs it prefers to put half the weight on each rather than all the weight on one.



Other kinds of weight penalty

- Sometimes it works better to penalize the absolute values of the weights.
 - This can make many weights exactly equal to zero which helps interpretation a lot.
- Sometimes it works better to use a weight penalty that has negligible effect on **large** weights.
 - This allows a few large weights.



Weight penalties vs weight constraints

- We usually penalize the squared value of each weight separately.
- Instead, we can put a constraint on the maximum squared length of the incoming weight vector of each unit.
 - If an update violates this constraint, we scale down the vector of incoming weights to the allowed length.
- Weight constraints have several advantages over weight penalties.
 - It's easier to set a sensible value.
 - They prevent hidden units getting stuck near zero.
 - They prevent weights exploding.
- When a unit hits it's limit, the effective weight penalty on all of it's weights is determined by the big gradients.
 - This is more effective than a fixed penalty at pushing irrelevant weights towards zero.

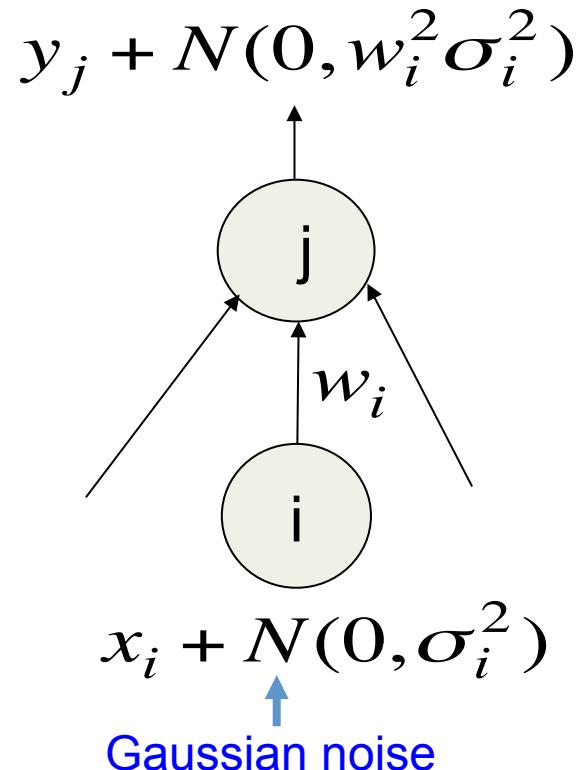
Neural Networks for Machine Learning

Lecture 9c Using noise as a regularizer

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

L2 weight-decay via noisy inputs

- Suppose we add Gaussian noise to the inputs.
 - The variance of the noise is amplified by the squared weight before going into the next layer.
- In a simple net with a linear output unit directly connected to the inputs, the amplified noise gets added to the output.
- This makes an additive contribution to the squared error.
 - So minimizing the squared error tends to minimize the squared weights when the inputs are noisy.



output on one case $\rightarrow y^{noisy} = \sum_i w_i x_i + \sum_i w_i \varepsilon_i$ where ε_i is sampled from $N(0, \sigma_i^2)$

$$E[(y^{noisy} - t)^2] = E\left[\left(y + \sum_i w_i \varepsilon_i - t\right)^2\right] = E\left[\left((y - t) + \sum_i w_i \varepsilon_i\right)^2\right]$$

$$= (y - t)^2 + E\left[2(y - t) \sum_i w_i \varepsilon_i\right] + E\left[\left(\sum_i w_i \varepsilon_i\right)^2\right]$$

$$= (y - t)^2 + E\left[\sum_i w_i^2 \varepsilon_i^2\right] \quad \begin{aligned} &\text{because } \varepsilon_i \text{ is independent of } \varepsilon_j \\ &\text{and } \varepsilon_i \text{ is independent of } (y - t) \end{aligned}$$

$$= (y - t)^2 + \sum_i w_i^2 \sigma_i^2 \quad \text{So } \sigma_i^2 \text{ is equivalent to an L2 penalty}$$

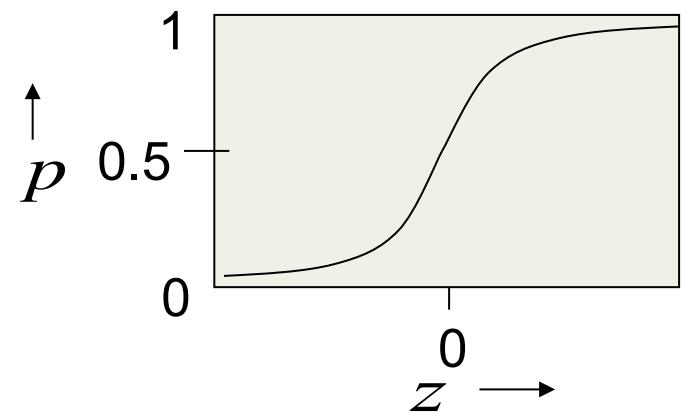
Noisy weights in more complex nets

- Adding Gaussian noise to the weights of a multilayer non-linear neural net is not exactly equivalent to using an L2 weight penalty.
 - It may work better, especially in recurrent networks.
 - Alex Graves' recurrent net that recognizes handwriting, works significantly better if noise is added to the weights.

Using noise in the activities as a regularizer

- Suppose we use backpropagation to train a multilayer neural net composed of logistic units.
 - What happens if we make the units binary and stochastic on the forward pass, but do the backward pass as if we had done the forward pass “properly”?
- It does worse on the training set and trains considerably slower.
 - But it does significantly better on the test set! (unpublished result).

$$p(s = 1) = \frac{1}{1 + e^{-z}}$$



Neural Networks for Machine Learning

Lecture 9d Introduction to the Bayesian Approach

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

The Bayesian framework

- The Bayesian framework assumes that we always have a prior distribution for everything.
 - The prior may be very vague.
 - When we see some data, we combine our prior distribution with a likelihood term to get a posterior distribution.
 - The likelihood term takes into account how probable the observed data is given the parameters of the model.
 - It favors parameter settings that make the data likely.
 - It fights the prior
 - With enough data the likelihood terms always wins.

A coin tossing example

- Suppose we know nothing about coins except that each tossing event produces a head with some unknown probability p and a tail with probability $1-p$.
 - Our model of a coin has one parameter, p .
- Suppose we observe 100 tosses and there are 53 heads.
What is p ?
- The frequentist answer (also called maximum likelihood): Pick the value of p that makes the observation of 53 heads and 47 tails most probable.
 - This value is $p=0.53$

A coin tossing example: the math

probability of
a particular
sequence
containing 53
heads and 47
tails.

$$\rightarrow P(D) = p^{53}(1-p)^{47}$$

$$\frac{dP(D)}{dp} = 53p^{52}(1-p)^{47} - 47p^{53}(1-p)^{46}$$

$$= \left(\frac{53}{p} - \frac{47}{1-p} \right) [p^{53}(1-p)^{47}]$$

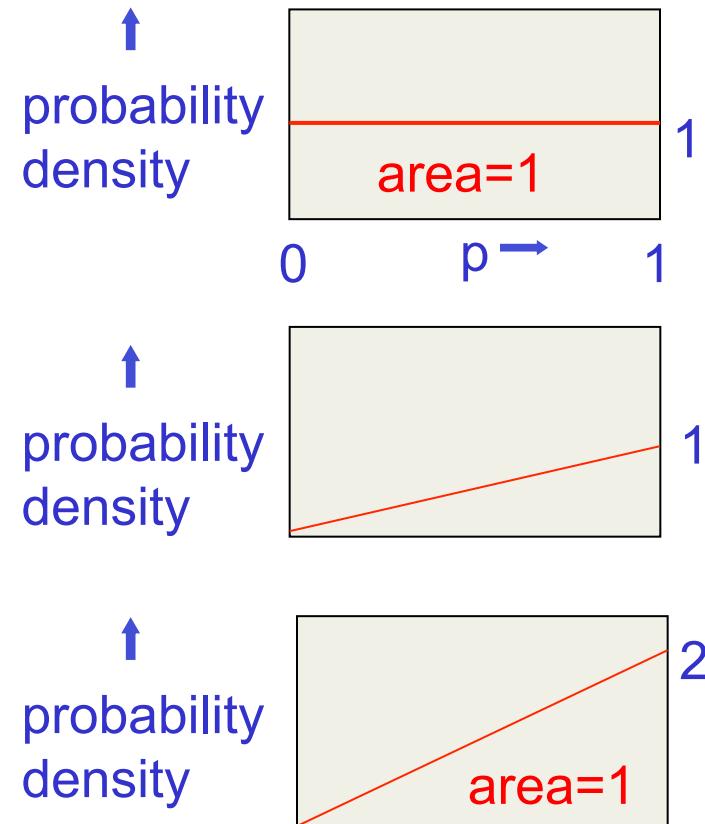
$$= 0 \text{ if } p = .53$$

Some problems with picking the parameters that are most likely to generate the data

- What if we only tossed the coin once and we got 1 head?
 - Is $p=1$ a sensible answer?
 - Surely $p=0.5$ is a much better answer.
- Is it reasonable to give a single answer?
 - If we don't have much data, we are unsure about p .
 - Our computations of probabilities will work much better if we take this uncertainty into account.

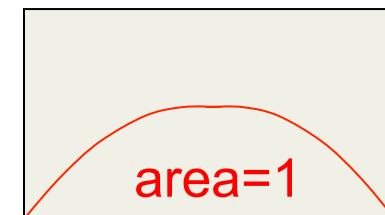
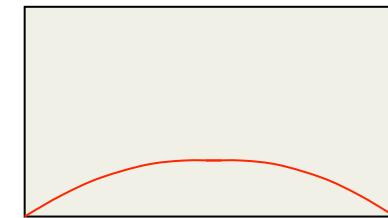
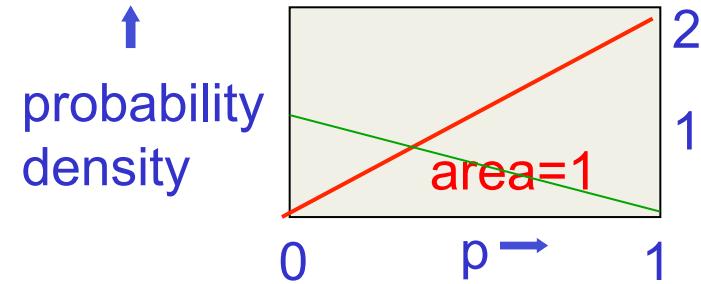
Using a distribution over parameter values

- Start with a prior distribution over p . In this case we used a uniform distribution.
- Multiply the prior probability of each parameter value by the probability of observing a head given that value.
- Then scale up all of the probability densities so that their integral comes to 1. This gives the posterior distribution.



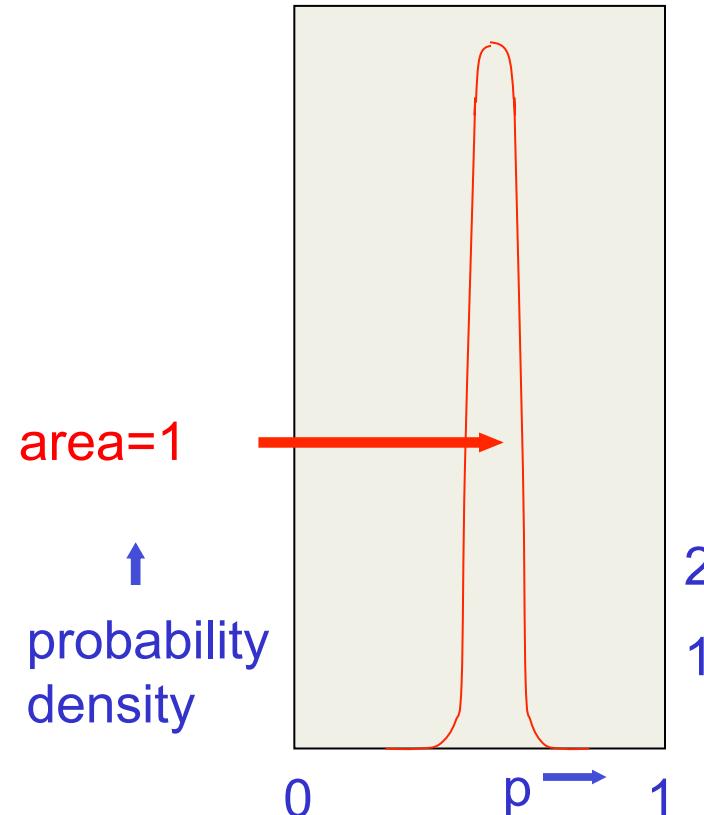
Lets do it again: Suppose we get a tail

- Start with a prior distribution over p .
- Multiply the prior probability of each parameter value by the probability of observing a **tail** given that value.
- Then renormalize to get the posterior distribution. **Look how sensible it is!**



Lets do it another 98 times

- After 53 heads and 47 tails we get a very sensible posterior distribution that has its peak at 0.53 (assuming a uniform prior).



Bayes Theorem

joint probability
↓

conditional
probability
↙

$$p(D)p(W|D) = p(D, W) = p(W)p(D|W)$$

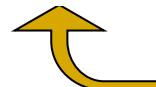
prior probability of
weight vector W



probability of observed
data given W

$$p(W|D) = \frac{p(W) p(D|W)}{p(D)}$$

posterior probability of
weight vector W given
training data D



$$\int_W p(W)p(D|W)$$

Neural Networks for Machine Learning

Lecture 9e

The Bayesian interpretation of weight decay

Geoffrey Hinton

Nitish Srivastava,

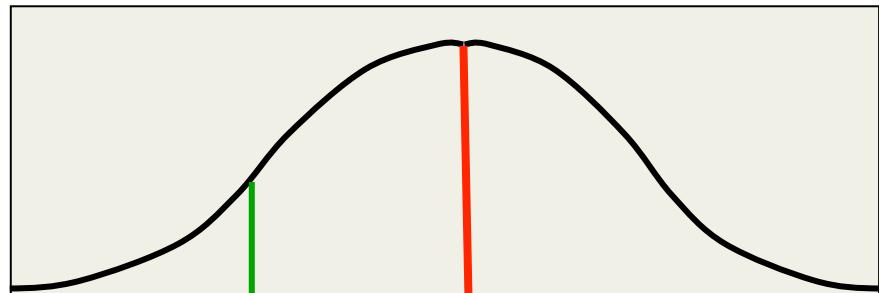
Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

Supervised Maximum Likelihood Learning

- Finding a weight vector that minimizes the squared residuals is equivalent to finding a weight vector that maximizes the log probability density of the correct answer.
- We assume the answer is generated by adding Gaussian noise to the output of the neural network.



t
correct
answer

y
model's
estimate of
most probable
value

Supervised Maximum Likelihood Learning

output of the net $\rightarrow y_c = f(\text{input}_c, W)$

probability density of the target value given the net's output plus Gaussian noise

$$p(t_c | y_c) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_c - y_c)^2}{2\sigma^2}}$$

Gaussian distribution centered at the net's output

Cost $\rightarrow -\log p(t_c | y_c) = k + \frac{(t_c - y_c)^2}{2\sigma^2}$

Minimizing squared error is the same as maximizing log prob under a Gaussian.

MAP: Maximum a Posteriori

- The proper Bayesian approach is to find the full posterior distribution over all possible weight vectors.
 - If we have more than a handful of weights this is hopelessly difficult for a non-linear net.
 - Bayesians have all sort of clever tricks for approximating this horrendous distribution.
- Suppose we just try to find the most probable weight vector.
 - We can find an optimum by starting with a random weight vector and then adjusting it in the direction that improves $p(W | D)$.
 - But it's only a local optimum.
- It is easier to work in the log domain. If we want to minimize a cost we use negative log probs

Why we maximize sums of log probabilities

- We want to maximize the **product** of the probabilities of the producing the target values on all the different training cases.
 - Assume the output errors on different cases, c , are independent.

$$p(D \mid W) = \prod_c p(t_c \mid W) = \prod_c p(t_c \mid f(\text{input}_c, W))$$

- Because the log function is monotonic, it does not change where the maxima are. So we can maximize **sums** of log probabilities

$$\log p(D \mid W) = \sum_c \log p(t_c \mid W)$$

MAP: Maximum a Posteriori

$$p(W | D) = p(W) p(D|W) / p(D)$$



$$Cost = -\log p(W | D) = -\log p(W) - \log p(D|W) + \log p(D)$$



log prob of
W under
the prior



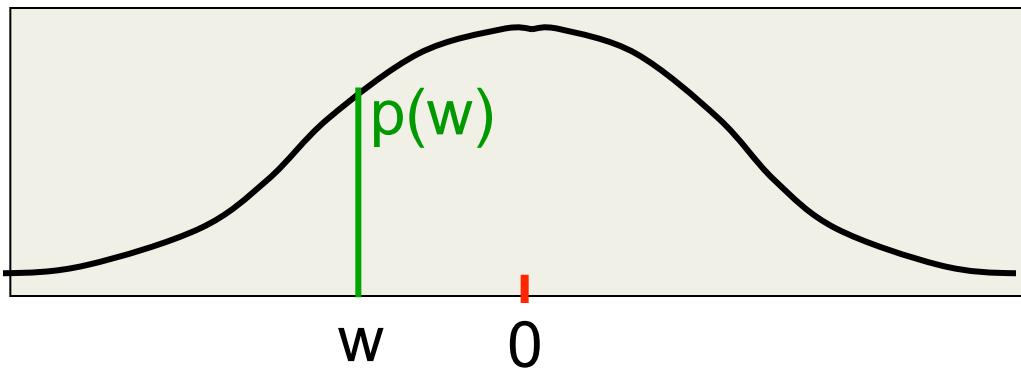
log prob
of target
values
given W



This is an integral over
all possible weight
vectors so it does not
depend on W

The log probability of a weight under its prior

- Minimizing the squared weights is equivalent to maximizing the log probability of the weights under a zero-mean Gaussian prior.



$$p(w) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{w^2}{2\sigma_w^2}}$$

$$-\log p(w) = \frac{w^2}{2\sigma_w^2} + k$$

The Bayesian interpretation of weight decay

$$-\log p(W|D) = -\log p(D|W) - \log p(W) + \log p(D)$$
$$C^* = \frac{1}{2\sigma_D^2} \sum_c (y_c - t_c)^2 + \frac{1}{2\sigma_W^2} \sum_i w_i^2$$

assuming that the model makes a Gaussian prediction

assuming a Gaussian prior for the weights

constant

$$C = E + \frac{\sigma_D^2}{\sigma_W^2} \sum_i w_i^2$$

So the correct value of the weight decay parameter is the ratio of two variances. It's not just an arbitrary hack.

Neural Networks for Machine Learning

Lecture 9f MacKay's quick and dirty method of fixing weight costs

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

Estimating the variance of the output noise

- After we have learned a model that minimizes the squared error, we can find the best value for the output noise.
 - The best value is the one that maximizes the probability of producing exactly the correct answers after adding Gaussian noise to the output produced by the neural net.
 - The best value is found by simply using the variance of the residual errors.

Estimating the variance of the Gaussian prior on the weights

- After learning a model with some initial choice of variance for the weight prior, we could do a dirty trick called “empirical Bayes”.
 - Set the variance of the Gaussian prior to be whatever makes the weights that the model learned most likely.
 - i.e. use the data itself to decide what your prior is!
 - This is done by simply fitting a zero-mean Gaussian to the one-dimensional distribution of the learned weight values.
 - We could easily learn different variances for different sets of weights.
- We don't need a validation set!

MacKay's quick and dirty method of choosing the ratio of the noise variance to the weight prior variance.

- Start with guesses for both the noise variance and the weight prior variance.
- While not yet bored
 - Do some learning using the ratio of the variances as the weight penalty coefficient.
 - Reset the noise variance to be the variance of the residual errors.
 - Reset the weight prior variance to be the variance of the distribution of the actual learned weights.
- Go back to the start of this loop.

Neural Networks for Machine Learning

Lecture 10a Why it helps to combine models

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

Combining networks: The bias-variance trade-off

- When the amount of training data is limited, we get overfitting.
 - Averaging the predictions of many different models is a good way to reduce overfitting.
 - It helps most when the models make very different predictions.
- For regression, the squared error can be decomposed into a “bias” term and a “variance” term.
 - The bias term is big if the model has too little capacity to fit the data.
 - The variance term is big if the model has so much capacity that it is good at fitting the sampling error in each particular training set.
- By averaging away the variance we can use individual models with high capacity. These models have high variance but low bias.

How the combined predictor compares with the individual predictors

- On any one test case, some individual predictors may be better than the combined predictor.
 - But different individual predictors will be better on different cases.
- If the individual predictors disagree a lot, the combined predictor is typically better than all of the individual predictors when we average over test cases.
 - So we should try to make the individual predictors disagree (without making them much worse individually).

Combining networks reduces variance

- We want to compare two expected squared errors: Pick a predictor at random versus use the average of all the predictors:

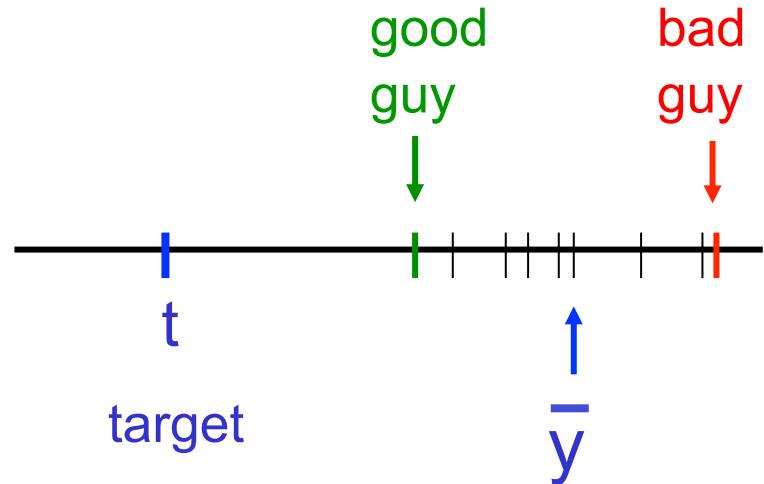
$$\bar{y} = \langle y_i \rangle_i = \frac{1}{N} \sum_{i=1}^N y_i \quad i \text{ is an index over the } N \text{ models}$$

$$\begin{aligned} \langle (t - y_i)^2 \rangle_i &= \langle ((t - \bar{y}) - (y_i - \bar{y}))^2 \rangle_i \\ &= \langle (t - \bar{y})^2 + (y_i - \bar{y})^2 - 2(t - \bar{y})(y_i - \bar{y}) \rangle_i \end{aligned} \quad \begin{matrix} \text{this term} \\ \text{vanishes} \end{matrix}$$

$$= (t - \bar{y})^2 + \langle (y_i - \bar{y})^2 \rangle_i - 2(t - \bar{y}) \langle (y_i - \bar{y}) \rangle_i$$

A picture

- The predictors that are further than average from t make bigger than average squared errors.
- The predictors that are nearer than average to t make smaller than average squared errors.
- The first effect dominates because **squares** work like that.
- Don't try averaging if you want to synchronize a bunch of clocks!
 - The noise is not Gaussian.

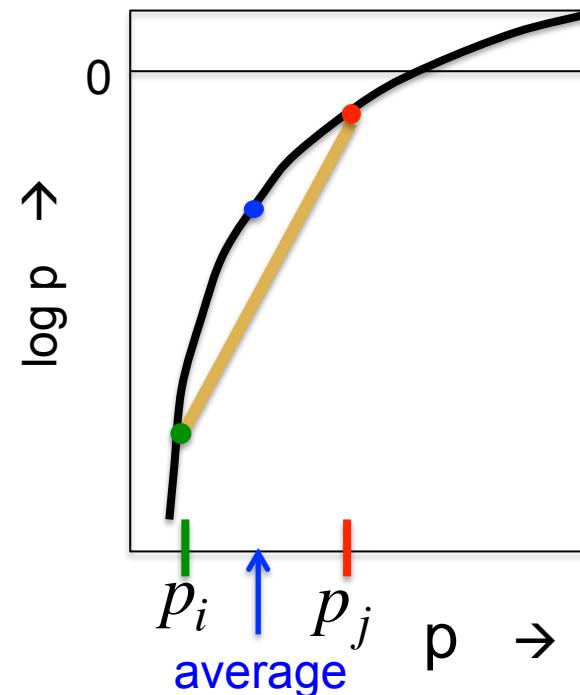


$$\frac{(\bar{y} - \varepsilon)^2 + (\bar{y} + \varepsilon)^2}{2} = \bar{y}^2 + \varepsilon^2$$

What about discrete distributions over class labels?

- Suppose that one model gives the correct label probability p_i and the other model gives it p_j
- Is it better to pick one model at random, or is it better to average the two probabilities?

$$\log\left(\frac{p_i + p_j}{2}\right) \geq \frac{\log p_i + \log p_j}{2}$$



Overview of ways to make predictors differ

- Rely on the learning algorithm getting stuck in different local optima.
 - A dubious hack
(but worth a try).
- Use lots of different kinds of models, including ones that are not neural networks.
 - Decision trees
 - Gaussian Process models
 - Support Vector Machines
 - and many others.
- For neural network models, make them different by using:
 - Different numbers of hidden layers.
 - Different numbers of units per layer.
 - Different types of unit.
 - Different types or strengths of weight penalty.
 - Different learning algorithms.

Making models differ by changing their training data

- **Bagging:** Train different models on different subsets of the data.
 - Bagging gets different training sets by using sampling with replacement:
a,b,c,d,e → a c c d d
 - Random forests use lots of different decision trees trained using bagging. They work well.
- We could use bagging with neural nets but its **very** expensive.
- **Boosting:** Train a sequence of low capacity models. Weight the training cases differently for each model in the sequence.
 - Boosting up-weights cases that previous models got wrong.
 - An early use of boosting was with neural nets for MNIST.
 - It focused the computational resources on modeling the tricky cases.

Neural Networks for Machine Learning

Lecture 10b Mixtures of Experts

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

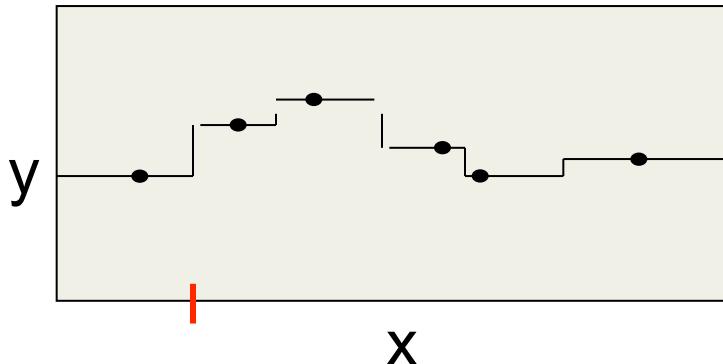
Mixtures of Experts

- Can we do better than just averaging models in a way that does not depend on the particular training case?
 - Maybe we can look at the input data for a particular case to help us decide which model to rely on.
 - This may allow particular models to specialize in a subset of the training cases.
 - They do not learn on cases for which they are not picked. So they can ignore stuff they are not good at modeling. Hurray for nerds!
- The key idea is to make each expert focus on predicting the right answer for the cases where it is already doing better than the other experts.
 - This causes specialization.

A spectrum of models

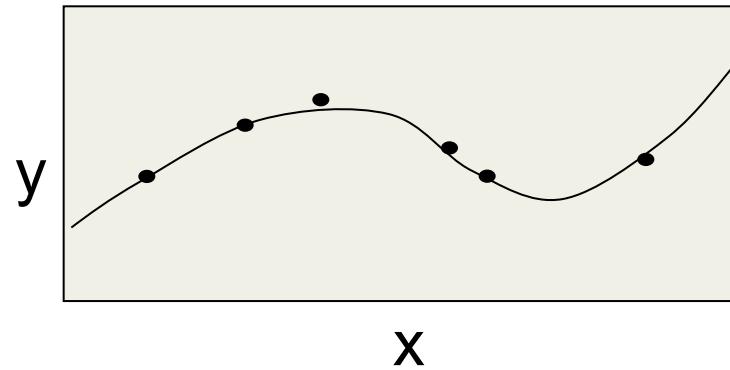
Very local models

- e.g. Nearest neighbors
- Very fast to fit
 - Just store training cases
- Local smoothing would obviously improve things.



Fully global models

- e.g. A polynomial
- May be slow to fit and also unstable.
 - Each parameter depends on all the data. Small changes to data can cause big changes to the fit.

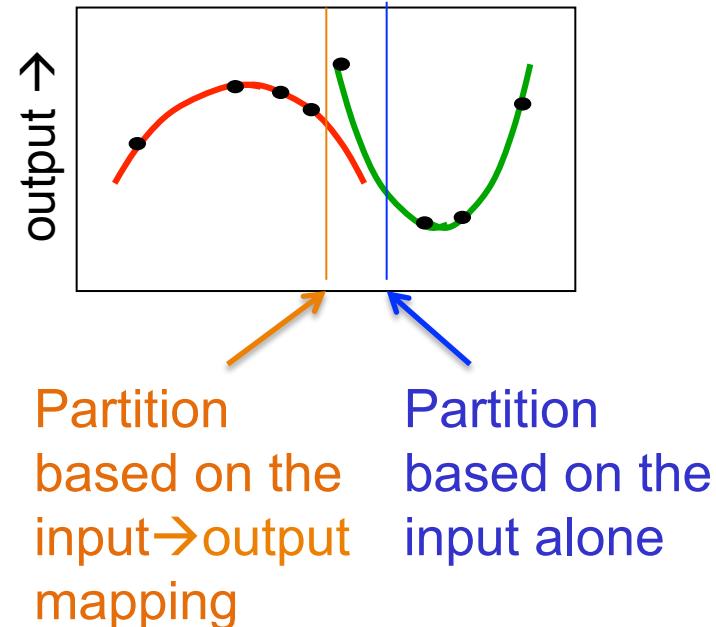


Multiple local models

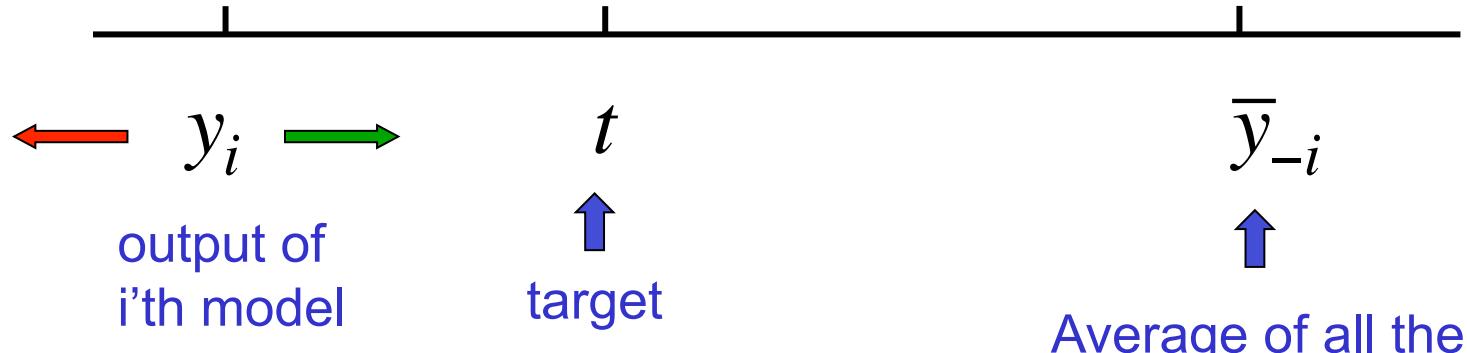
- Instead of using a single global model or lots of very local models, use several models of intermediate complexity.
 - Good if the dataset contains several different regimes which have different relationships between input and output.
 - e.g. financial data which depends on the state of the economy.
- But how do we partition the dataset into regimes?

Partitioning based on input alone versus partitioning based on the input-output relationship

- We need to cluster the training cases into subsets, one for each local model.
 - The aim of the clustering is NOT to find clusters of similar input vectors.
 - We want each cluster to have a relationship between input and output that can be well-modeled by one local model.



A picture of why averaging models during training causes cooperation not specialization



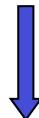
Do we really want to move the output of model i away from the target value?

Average of all the other predictors

An error function that encourages cooperation

- If we want to encourage cooperation, we compare the average of all the predictors with the target and train to reduce the discrepancy.
 - This can overfit badly. It makes the model much more powerful than training each predictor separately.

Average of all
the predictors



$$E = (t - \langle y_i \rangle_i)^2$$

An error function that encourages specialization

- If we want to encourage specialization we compare each predictor separately with the target.
- We also use a “manager” to determine the probability of picking each expert.
 - Most experts end up ignoring most targets

probability of the manager picking expert i for this case

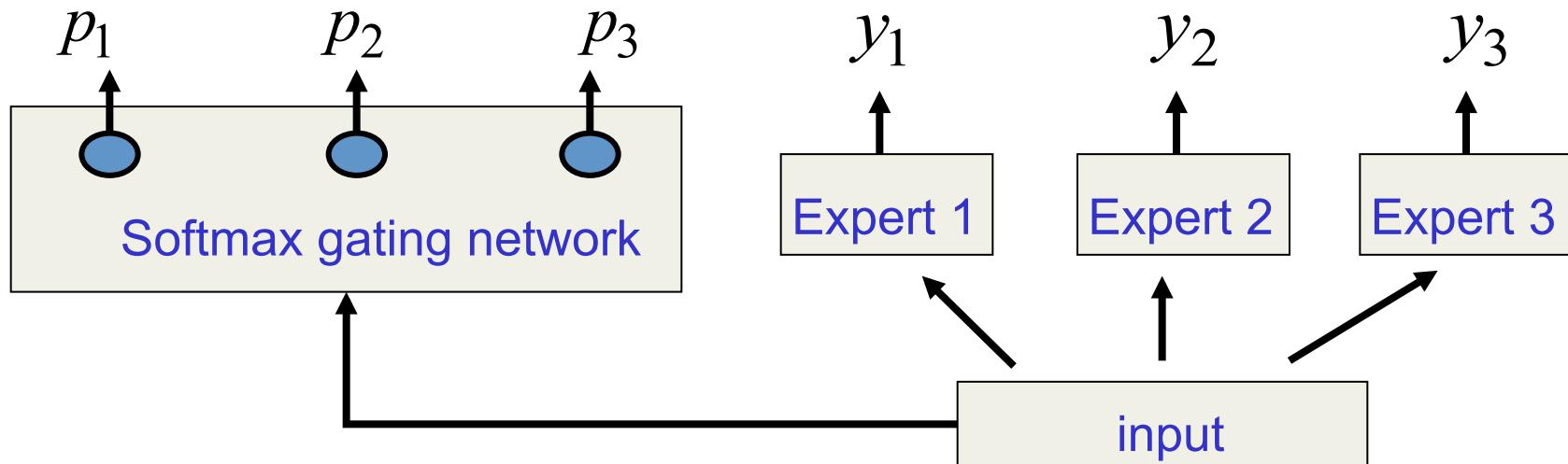


$$E = \langle p_i(t - y_i)^2 \rangle_i$$

The mixture of experts architecture (almost)

A simple cost function : $E = \sum_i p_i(t - y_i)^2$

There is a better cost function based on a mixture model.



The derivatives of the simple cost function

- If we differentiate w.r.t. the outputs of the experts we get a signal for training each expert.
- If we differentiate w.r.t. the outputs of the gating network we get a signal for training the gating net.

– We want to raise p for all experts that give less than the average squared error of all the experts (weighted by p)

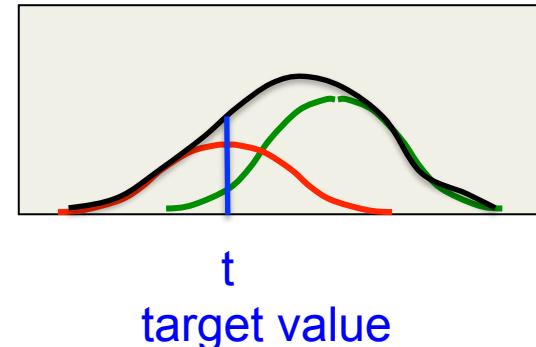
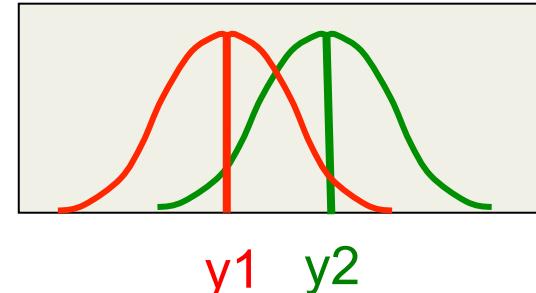
$$p_i = \frac{e^{x_i}}{\sum_j e^{x_j}}, \quad E = \sum_i p_i(t - y_i)^2,$$

$$\frac{\partial E}{\partial y_i} = p_i(t - y_i)$$

$$\frac{\partial E}{\partial x_i} = p_i((t - y_i)^2 - E)$$

A better cost function for mixtures of experts (Jacobs, Jordan, Nowlan & Hinton, 1991)

- Think of each expert as making a prediction that is a Gaussian distribution around its output (**with variance 1**).
- Think of the manager as deciding on a scale for each of these Gaussians. The scale is called a “mixing proportion”. e.g **{0.4 0.6}**
- Maximize the log probability of the target value under this mixture of Gaussians model i.e. the sum of the two scaled Gaussians.



The probability of the target under a mixture of Gaussians

mixing proportion assigned to expert i
for case c by the gating network

$$p(t^c | MoE) = \sum_i p_i^c \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2} (t^c - y_i^c)^2}$$

prob. of target value on case c given the mixture.

output of expert i

normalization term for a Gaussian with $\sigma^2 = 1$

Neural Networks for Machine Learning

Lecture 10c The idea of full Bayesian learning

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

Full Bayesian Learning

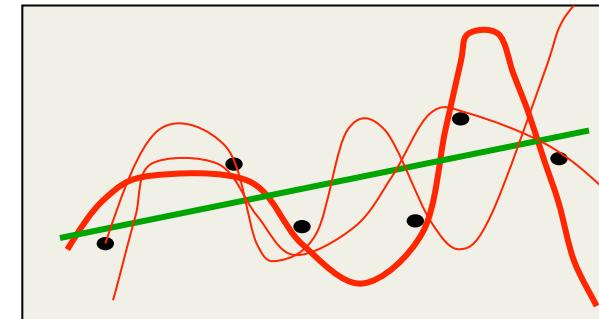
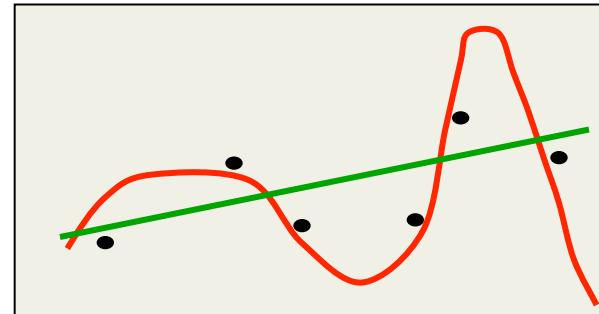
- Instead of trying to find the best single setting of the parameters (as in Maximum Likelihood or MAP) compute the full posterior distribution over all possible parameter settings.
 - This is extremely computationally intensive for all but the simplest models (its feasible for a biased coin).
- To make predictions, let each different setting of the parameters make its own prediction and then combine all these predictions by weighting each of them by the posterior probability of that setting of the parameters.
 - This is also very computationally intensive.
- The full Bayesian approach allows us to use complicated models even when we do not have much data.

Overfitting: A frequentist illusion?

- If you do not have much data, you should use a simple model, because a complex one will overfit.
 - This is true.
 - But only if you assume that fitting a model means choosing a single best setting of the parameters.
- If you use the full posterior distribution over parameter settings, overfitting disappears.
 - When there is very little data, you get very vague predictions because many different parameters settings have significant posterior probability.

A classic example of overfitting

- Which model do you believe?
 - The complicated model fits the data better.
 - But it is not economical and it makes silly predictions.
- But what if we start with a reasonable prior over all fifth-order polynomials and use the full posterior distribution.
 - Now we get vague and sensible predictions.
- There is no reason why the amount of data should influence our prior beliefs about the complexity of the model.



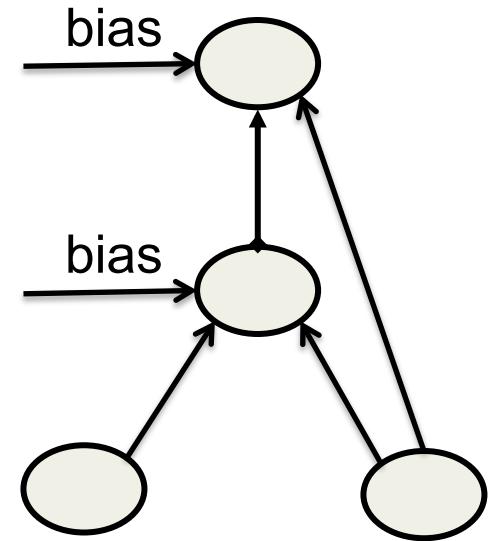
Approximating full Bayesian learning in a neural net

- If the neural net only has a few parameters we could put a grid over the parameter space and evaluate $p(W | D)$ at each grid-point.
 - This is expensive, but it does not involve any gradient descent and there are no local optimum issues.
- After evaluating each grid point we use all of them to make predictions on test data
 - This is also expensive, but it works much better than ML learning when the posterior is vague or multimodal (this happens when data is scarce).

$$p(t_{test} | \text{input}_{test}) = \sum_{g \in \text{grid}} p(W_g | D) \ p(t_{test} | \text{input}_{test}, W_g)$$

An example of full Bayesian learning

- Allow each of the 6 weights or biases to have the 9 possible values -2, -1.5, -1, -0.5, 0 ,0.5, 1, 1.5, 2
 - There are 9^6 grid-points in parameter space
- For each grid-point compute the probability of the observed outputs of all the training cases.
- Multiply the prior for each grid-point by the likelihood term and renormalize to get the posterior probability for each grid-point.
- Make predictions by using the posterior probabilities to average the predictions made by the different grid-points.



A neural net with 2 inputs, 1 output and 6 parameters

Neural Networks for Machine Learning

Lecture 10d Making full Bayesian learning practical

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

What can we do if there are too many parameters for a grid?

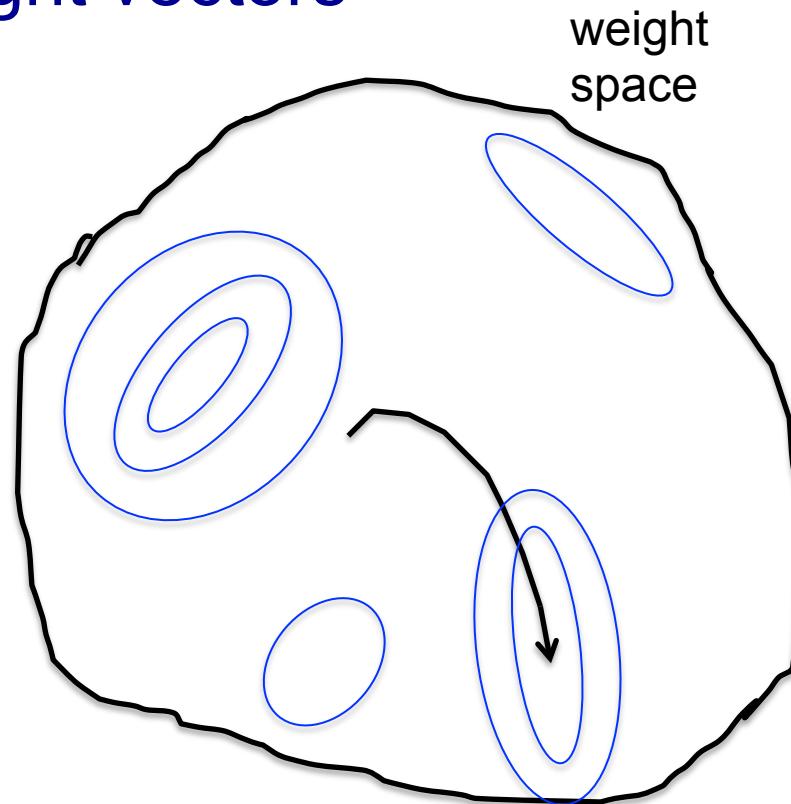
- The number of grid points is exponential in the number of parameters.
 - So we cannot deal with more than a few parameters using a grid.
- If there is enough data to make most parameter vectors very unlikely, only a tiny fraction of the grid points make a significant contribution to the predictions.
 - Maybe we can just evaluate this tiny fraction
- Idea: It might be good enough to just sample weight vectors according to their posterior probabilities.

$$p(y_{test} | \text{input}_{test}, D) = \sum_i p(W_i | D) \ p(y_{test} | \text{input}_{test}, W_i)$$

↑
Sample weight vectors
with this probability

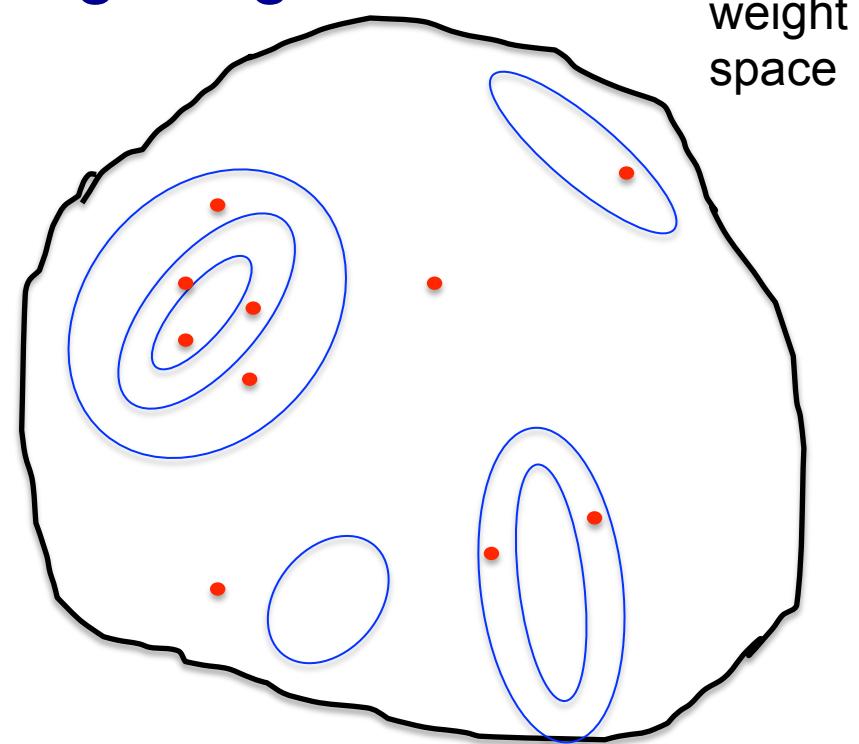
Sampling weight vectors

- In standard backpropagation we keep moving the weights in the direction that decreases the cost.
 - i.e. the direction that increases the log likelihood plus the log prior, summed over all training cases.
 - Eventually, the weights settle into a local minimum or get stuck on a plateau or just move so slowly that we run out of patience.



One method for sampling weight vectors

- Suppose we add some Gaussian noise to the weight vector after each update.
 - So the weight vector never settles down.
 - It keeps wandering around, but it tends to prefer low cost regions of the weight space.
 - Can we say anything about how often it will visit each possible setting of the weights?



Save the weights after every 10,000 steps.

The wonderful property of Markov Chain Monte Carlo

- Amazing fact: If we use just the right amount of noise, and if we let the weight vector wander around for long enough before we take a sample, we will get an unbiased sample from the true posterior over weight vectors.
 - This is called a “Markov Chain Monte Carlo” method.
 - MCMC makes it feasible to use full Bayesian learning with thousands of parameters.
- There are related MCMC methods that are more complicated but more efficient:
 - We don’t need to let the weights wander around for so long before we get samples from the posterior.

Full Bayesian learning with mini-batches

- If we compute the gradient of the cost function on a random mini-batch we will get an unbiased estimate with sampling noise.
 - Maybe we can use the sampling noise to provide the noise that an MCMC method needs!
- Ahn, Korattikara &Welling (ICML 2012) showed how to do this fairly efficiently.
 - So full Bayesian learning is now possible with lots of parameters.

Neural Networks for Machine Learning

Lecture 10e

Dropout: an efficient way to combine neural nets

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

Two ways to average models

- MIXTURE: We can combine models by averaging their output probabilities:
- PRODUCT: We can combine models by taking the geometric means of their output probabilities:

Model A: .3 .2 .5

Model B: .1 .8 .1

Combined .2 .5 .3

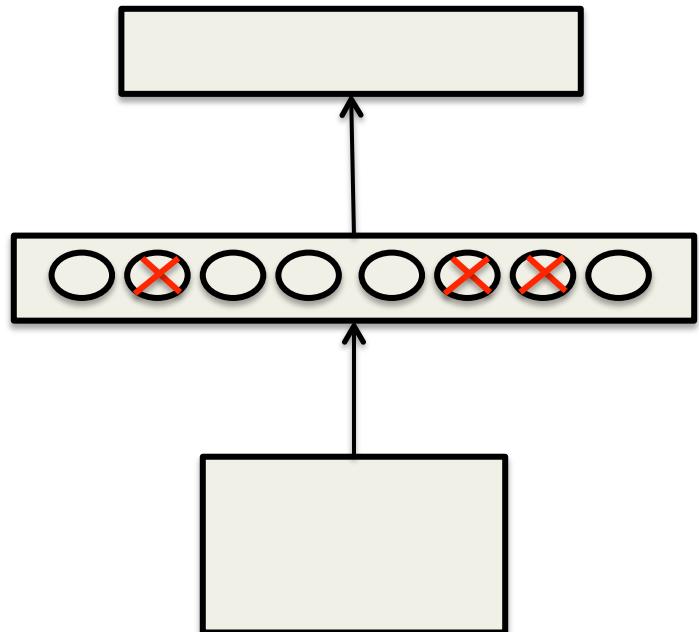
Model A: .3 .2 .5

Model B: .1 .8 .1

Combined $\sqrt{.03}$ $\sqrt{.16}$ $\sqrt{.05}$ /sum

Dropout: An efficient way to average many large neural nets (<http://arxiv.org/abs/1207.0580>)

- Consider a neural net with one hidden layer.
- Each time we present a training example, we randomly omit each hidden unit with probability 0.5.
- So we are randomly sampling from 2^H different architectures.
 - All architectures share weights.



Dropout as a form of model averaging

- We sample from 2^H models. So only a few of the models ever get trained, and they only get one training example.
 - This is as extreme as bagging can get.
- The sharing of the weights means that every model is very strongly regularized.
 - It's a much better regularizer than L2 or L1 penalties that pull the weights towards zero.

But what do we do at test time?

- We could sample many different architectures and take the geometric mean of their output distributions.
- It better to use all of the hidden units, but to halve their outgoing weights.
 - This exactly computes the geometric mean of the predictions of all 2^H models.

What if we have more hidden layers?

- Use dropout of 0.5 in every layer.
- At test time, use the “mean net” that has all the outgoing weights halved.
 - This is not exactly the same as averaging all the separate dropped out models, but it’s a pretty good approximation, and its fast.
- Alternatively, run the stochastic model several times on the same input.
 - This gives us an idea of the uncertainty in the answer.

What about the input layer?

- It helps to use dropout there too, but with a higher probability of keeping an input unit.
 - This trick is already used by the “denoising autoencoders” developed by Pascal Vincent, Hugo Larochelle and Yoshua Bengio.

How well does dropout work?

- The record breaking object recognition net developed by Alex Krizhevsky (see lecture 5) uses dropout and it helps a lot.
- If your deep neural net is significantly overfitting, dropout will usually reduce the number of errors by a lot.
 - Any net that uses “early stopping” can do better by using dropout (at the cost of taking quite a lot longer to train).
- If your deep neural net is not overfitting you should be using a bigger one!

Another way to think about dropout

- If a hidden unit knows which other hidden units are present, it can co-adapt to them on the training data.
 - But complex co-adaptations are likely to go wrong on new test data.
 - Big, complex conspiracies are not robust.
- If a hidden unit has to work well with combinatorially many sets of co-workers, it is more likely to do something that is individually useful.
 - But it will also tend to do something that is marginally useful given what its co-workers achieve.

Neural Networks for Machine Learning

Lecture 11a Hopfield Nets

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

Hopfield Nets

- A Hopfield net is composed of binary threshold units with recurrent connections between them.
- Recurrent networks of non-linear units are generally very hard to analyze. They can behave in many different ways:
 - Settle to a stable state
 - Oscillate
 - Follow chaotic trajectories that cannot be predicted far into the future.
- But John Hopfield (and others) realized that if the connections are **symmetric**, there is a global energy function.
 - Each binary “configuration” of the whole network has an energy.
 - The binary threshold decision rule causes the network to settle to a minimum of this energy function.

The energy function

- The global energy is the sum of many contributions. Each contribution depends on **one connection weight** and the binary states of **two** neurons:

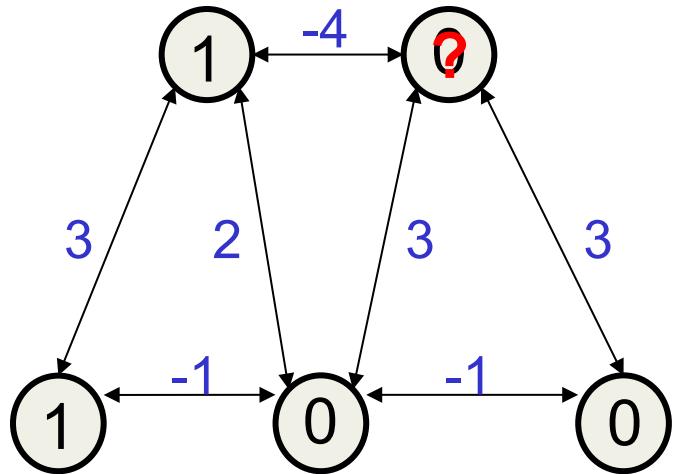
$$E = - \sum_i s_i b_i - \sum_{i < j} s_i s_j w_{ij}$$

- This simple **quadratic** energy function makes it possible for each unit to compute **locally** how its state affects the global energy:

$$\text{Energy gap} = \Delta E_i = E(s_i = 0) - E(s_i = 1) = b_i + \sum_j s_j w_{ij}$$

Settling to an energy minimum

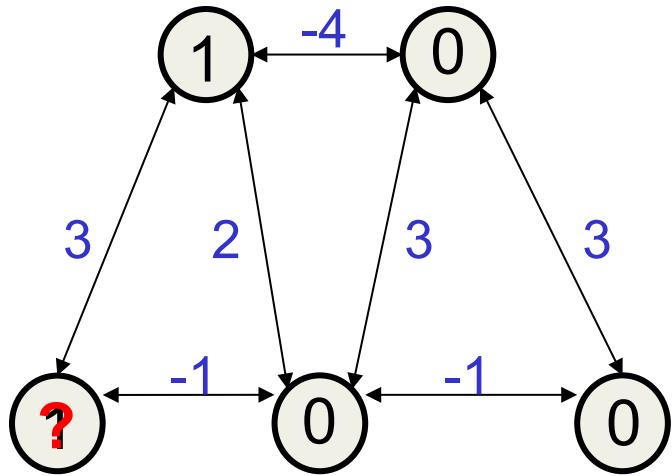
- To find an energy minimum in this net, start from a random state and then update units **one at a time** in random order.
 - Update each unit to whichever of its two states gives the lowest global energy.
 - i.e. use binary threshold units.



- $E = \text{goodness} = 3$

Settling to an energy minimum

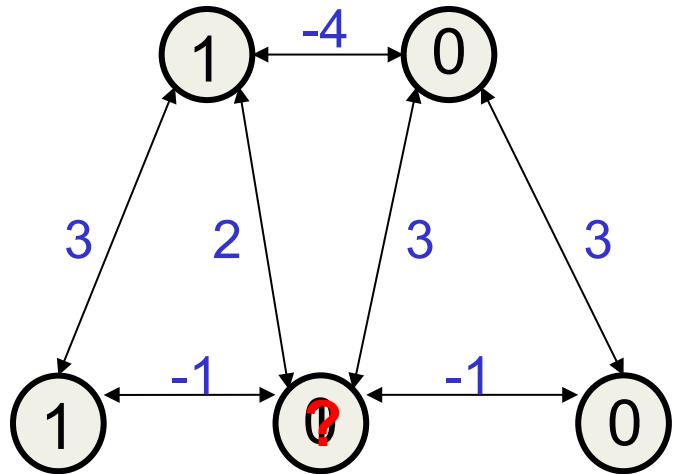
- To find an energy minimum in this net, start from a random state and then update units **one at a time** in random order.
 - Update each unit to whichever of its two states gives the lowest global energy.
 - i.e. use binary threshold units.



- $E = \text{goodness} = 3$

Settling to an energy minimum

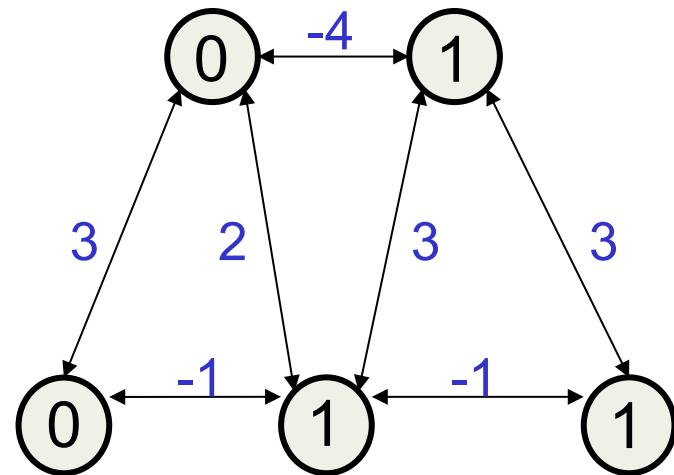
- To find an energy minimum in this net, start from a random state and then update units **one at a time** in random order.
 - Update each unit to whichever of its two states gives the lowest global energy.
 - i.e. use binary threshold units.



$$- E = \text{goodness} = 4$$

A deeper energy minimum

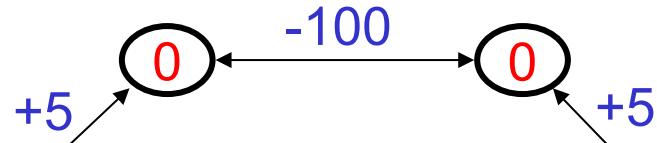
- The net has two triangles in which the three units mostly support each other.
 - Each triangle mostly hates the other triangle.
- The triangle on the left differs from the one on the right by having a weight of 2 where the other one has a weight of 3.
 - So turning on the units in the triangle on the right gives the deepest minimum.



$$- E = \text{goodness} = 5$$

Why do the decisions need to be sequential?

- If units make **simultaneous** decisions the energy could go up.
- With simultaneous parallel updating we can get oscillations.
 - They always have a period of 2.
- If the updates occur in parallel but with random timing, the oscillations are usually destroyed.



At the next parallel step, both units will turn on. This has very high energy, so then they will both turn off again.

A neat way to make use of this type of computation

- Hopfield (1982) proposed that memories could be energy minima of a neural net.
 - The binary threshold decision rule can then be used to “clean up” incomplete or corrupted memories.
- The idea of memories as energy minima was proposed by I. A. Richards in 1924 in “Principles of Literary Criticism”.
- Using energy minima to represent memories gives a content-addressable memory:
 - An item can be accessed by just knowing part of its content.
 - This was really amazing in the year 16 BG.
 - It is robust against hardware damage.
 - It’s like reconstructing a dinosaur from a few bones.

Storing memories in a Hopfield net

- If we use activities of 1 and -1, we can store a binary state vector by incrementing the weight between any two units by the product of their activities.
 - We treat biases as weights from a permanently on unit.
- With states of 0 and 1 the rule is slightly more complicated.

$$\Delta w_{ij} = s_i s_j$$

This is a very simple rule that is not error-driven. That is both its strength and its weakness

$$\Delta w_{ij} = 4 \left(s_i - \frac{1}{2} \right) \left(s_j - \frac{1}{2} \right)$$

Neural Networks for Machine Learning

Lecture 11b

Dealing with spurious minima in Hopfield Nets

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

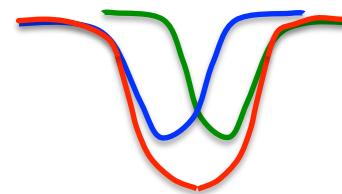
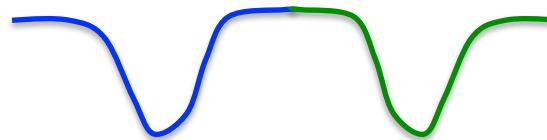
Abdel-rahman Mohamed

The storage capacity of a Hopfield net

- Using Hopfield's storage rule the capacity of a totally connected net with N units is only about $0.15N$ memories.
 - At N bits per memory this is only $0.15 N^2$ bits.
 - This does not make efficient use of the bits required to store the weights.
- The net has N^2 weights and biases.
- After storing M memories, each connection weight has an integer value in the range $[-M, M]$.
- So the number of bits required to store the weights and biases is: $N^2 \log(2M + 1)$

Spurious minima limit capacity

- Each time we memorize a configuration, we hope to create a new energy minimum.
 - But what if two nearby minima merge to create a minimum at an intermediate location?
 - This limits the capacity of a Hopfield net.



The state space is the corners of a hypercube. Showing it as a 1-D continuous space is a misrepresentation.

Avoiding spurious minima by unlearning

- Hopfield, Feinstein and Palmer suggested the following strategy:
 - Let the net settle from a random initial state and then do **unlearning**.
 - This will get rid of deep, spurious minima and increase memory capacity.
- They showed that this worked.
 - But they had no analysis.
- Crick and Mitchison proposed unlearning as a model of what dreams are for.
 - That's why you don't remember them (unless you wake up during the dream)
- But how much unlearning should we do?
 - Can we derive unlearning as the right way to minimize some cost function?

Increasing the capacity of a Hopfield net

- Physicists love the idea that the math they already know might explain how the brain works.
 - Many papers were published in physics journals about Hopfield nets and their storage capacity.
- Eventually, Elizabeth Gardiner figured out that there was a much better storage rule that uses the full capacity of the weights.
 - Instead of trying to store vectors in one shot, cycle through the training set many times.
 - Use the perceptron convergence procedure to train each unit to have the correct state given the states of all the other units in that vector.
 - Statisticians call this technique “pseudo-likelihood”.

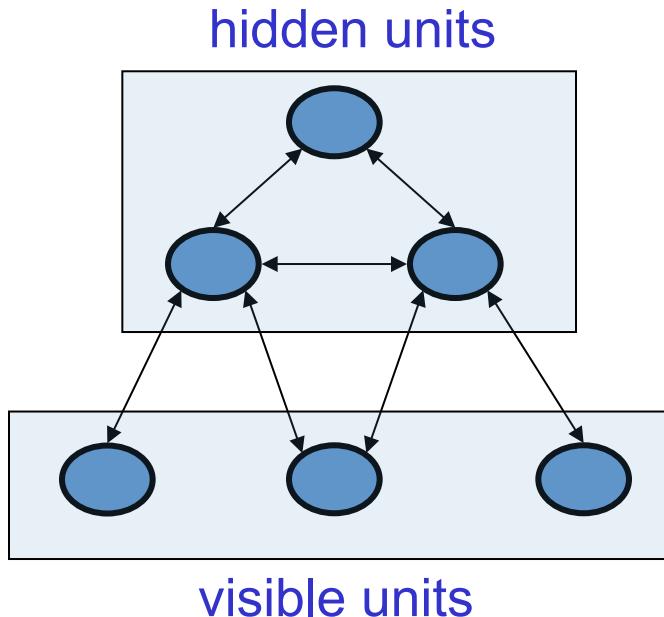
Neural Networks for Machine Learning

Lecture 11c Hopfield Nets with hidden units

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

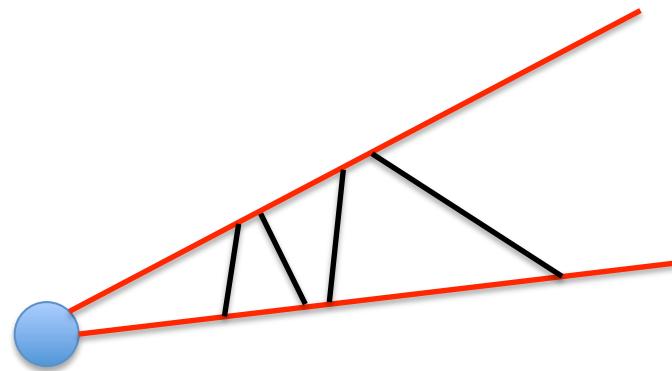
A different computational role for Hopfield nets

- Instead of using the net to store memories, use it to construct interpretations of sensory input.
 - The input is represented by the visible units.
 - The interpretation is represented by the states of the hidden units.
 - The badness of the interpretation is represented by the energy.



What can we infer about 3-D edges from 2-D lines in an image?

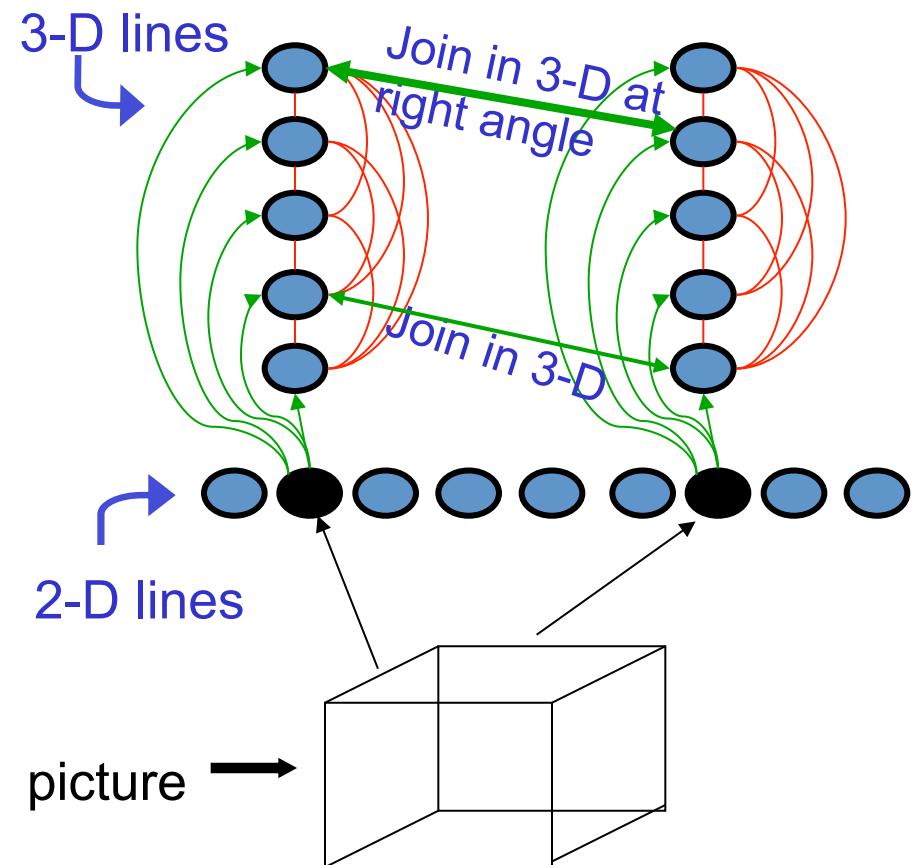
- A 2-D line in an image could have been caused by many different 3-D edges in the world.
- If we assume it's a straight 3-D edge, the information that has been lost in the image is the 3-D depth of each end of the 2-D line.
 - So there is a family of 3-D edges that all correspond to the same 2-D line.



You can only see one of these 3-D edges at a time because they occlude one another.

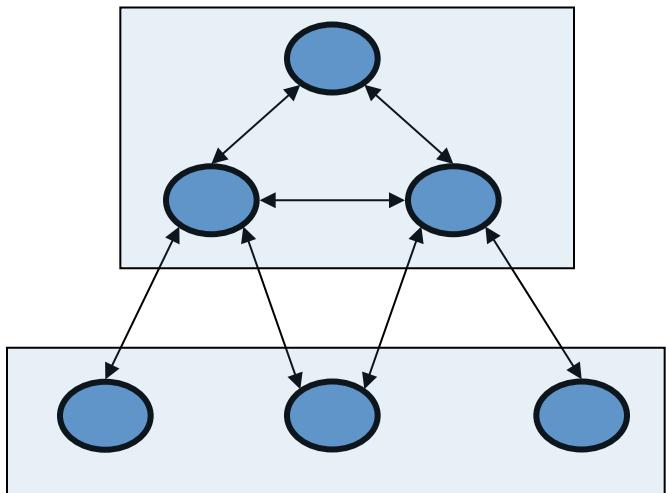
An example: Interpreting a line drawing

- Use one “2-D line” unit for each possible line in the picture.
 - Any particular picture will only activate a very small subset of the line units.
- Use one “3-D line” unit for each possible 3-D line in the scene.
 - Each 2-D line unit could be the projection of many possible 3-D lines. Make these 3-D lines **compete**.
- Make 3-D lines **support** each other if they join in 3-D.
- Make them **strongly support** each other if they join at right angles.



Two difficult computational issues

- Using the states of the hidden units to represent an interpretation of the input raises two difficult issues:
 - Search (lecture 11) How do we avoid getting trapped in poor local minima of the energy function?
 - Poor minima represent sub-optimal interpretations.
 - Learning (lecture 12) How do we learn the weights on the connections to the hidden units and between the hidden units?



Neural Networks for Machine Learning

Lecture 11d

Using stochastic units to improve search

Geoffrey Hinton

Nitish Srivastava,

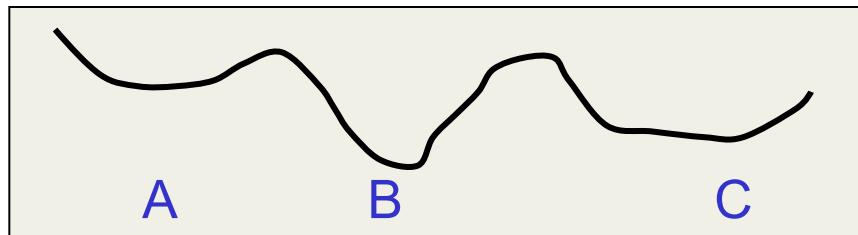
Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

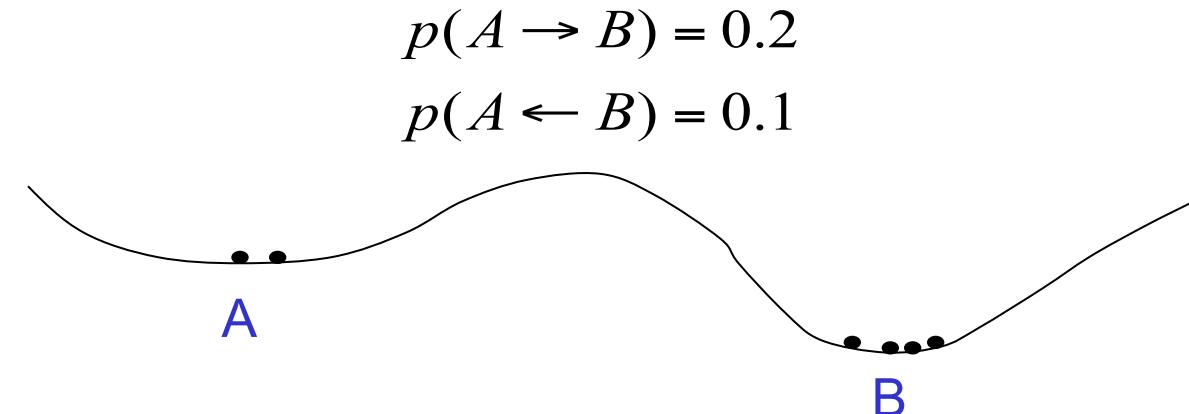
Noisy networks find better energy minima

- A Hopfield net always makes decisions that reduce the energy.
 - This makes it impossible to escape from local minima.
- We can use random noise to escape from poor minima.
 - Start with a lot of noise so its easy to cross energy barriers.
 - Slowly reduce the noise so that the system ends up in a deep minimum. This is “simulated annealing” (Kirkpatrick et.al. 1981)

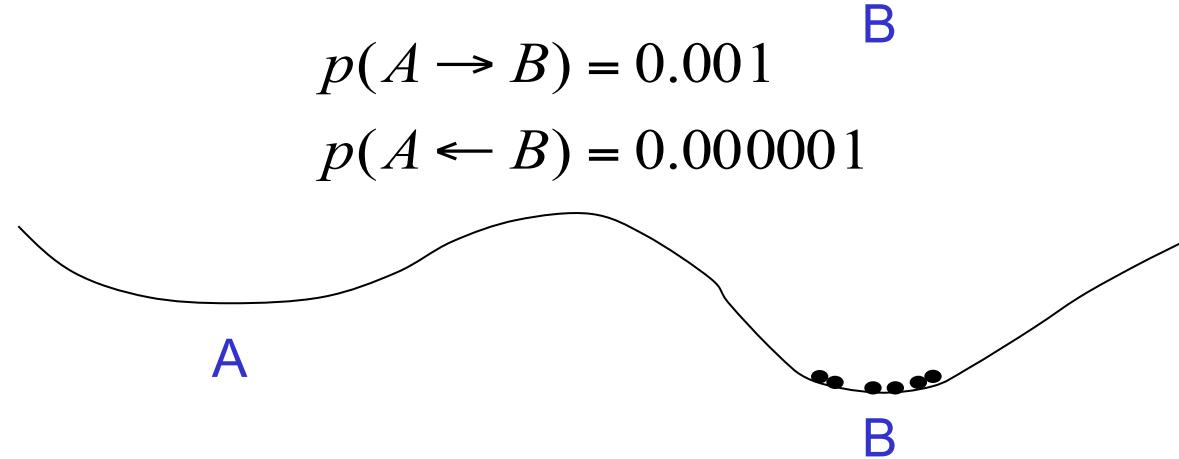


How temperature affects transition probabilities

High temperature
transition
probabilities



Low temperature
transition
probabilities



Stochastic binary units

- Replace the binary threshold units by binary stochastic units that make biased random decisions.
 - The “temperature” controls the amount of noise
 - Raising the noise level is equivalent to decreasing all the energy gaps between configurations.

$$p(s_i=1) = \frac{1}{1 + e^{-\Delta E_i/T}} \quad \text{← temperature}$$

$$\text{Energy gap} = \Delta E_i = E(s_i = 0) - E(s_i = 1) = b_i + \sum_j s_j w_{ij}$$

Simulated annealing is a distraction

- Simulated annealing is a powerful method for improving searches that get stuck in local optima.
- It was one of the ideas that led to Boltzmann machines.
- But it's a big distraction from the main ideas behind Boltzmann machines.
 - So it will not be covered in this course.
 - From now on, we will use binary stochastic units that have a temperature of 1.

Thermal equilibrium at a temperature of 1

- Thermal equilibrium is a difficult concept!
 - Reaching thermal equilibrium does not mean that the system has settled down into the lowest energy configuration.
 - The thing that settles down is the **probability distribution** over configurations.
 - This settles to the stationary distribution.
- There is a nice intuitive way to think about thermal equilibrium:
 - Imagine a huge ensemble of systems that all have exactly the same energy function.
 - The probability of a configuration is just the fraction of the systems that have that configuration.

Approaching thermal equilibrium

- We start with any distribution we like over all the identical systems.
 - We could start with all the systems in the same configuration.
 - Or with an equal number of systems in each possible configuration.
- Then we keep applying our stochastic update rule to pick the next configuration for each individual system.
- After running the systems stochastically in the right way, we may eventually reach a situation where the fraction of systems in each configuration remains constant.
 - This is the stationary distribution that physicists call thermal equilibrium.
 - Any given system keeps changing its configuration, but the fraction of systems in each configuration does not change.

An analogy

- Imagine a casino in Las Vegas that is full of card dealers (we need many more than $52!$ of them).
- We start with all the card packs in standard order and then the dealers all start shuffling their packs.
 - After a few time steps, the king of spades still has a good chance of being next to the queen of spades. The packs have not yet forgotten where they started.
 - After prolonged shuffling, the packs will have forgotten where they started. There will be an equal number of packs in each of the $52!$ possible orders.
 - Once equilibrium has been reached, the number of packs that leave a configuration at each time step will be equal to the number that enter the configuration.
- The only thing wrong with this analogy is that all the configurations have equal energy, so they all end up with the same probability.

Neural Networks for Machine Learning

Lecture 11e

How a Boltzmann Machine models data

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

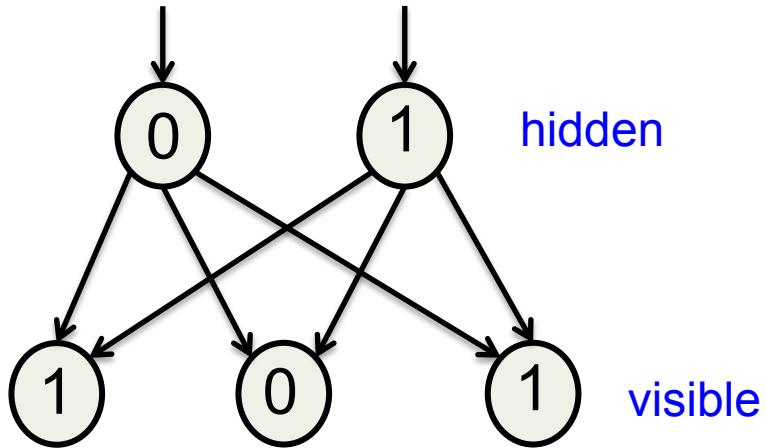
Modeling binary data

- Given a training set of binary vectors, fit a model that will assign a probability to every possible binary vector.
 - This is useful for deciding if other binary vectors come from the same distribution (e.g. documents represented by binary features that represents the occurrence of a particular word).
 - It can be used for monitoring complex systems to detect unusual behavior.
 - If we have models of several different distributions it can be used to compute the posterior probability that a particular distribution produced the observed data.

$$p(\text{Model } i \mid \text{data}) = \frac{p(\text{data} \mid \text{Model } i)}{\sum_j p(\text{data} \mid \text{Model } j)}$$

How a causal model generates data

- In a causal model we generate data in two sequential steps:
 - First pick the hidden states from their prior distribution.
 - Then pick the visible states from their conditional distribution given the hidden states.
- The probability of generating a visible vector, v , is computed by summing over all possible hidden states. Each hidden state is an “explanation” of v .



$$p(v) = \sum_h p(h)p(v | h)$$

How a Boltzmann Machine generates data

- It is **not** a causal generative model.
- Instead, everything is defined in terms of the energies of joint configurations of the visible and hidden units.
- The energies of joint configurations are related to their probabilities in two ways.
 - We can simply define the probability to be $p(\mathbf{v}, \mathbf{h}) \propto e^{-E(\mathbf{v}, \mathbf{h})}$
 - Alternatively, we can define the probability to be the probability of finding the network in that joint configuration after we have updated all of the stochastic binary units many times.
- These two definitions agree.

The Energy of a joint configuration

$$-E(\mathbf{v}, \mathbf{h}) = \sum_{i \in vis} v_i b_i + \sum_{k \in hid} h_k b_k + \sum_{i < j} v_i v_j w_{ij} + \sum_{i, k} v_i h_k w_{ik} + \sum_{k < l} h_k h_l w_{kl}$$

binary state of unit i in \mathbf{v}

bias of unit k

Energy with configuration \mathbf{v} on the visible units and \mathbf{h} on the hidden units

indexes every non-identical pair of i and j once

weight between visible unit i and hidden unit k

Using energies to define probabilities

- The probability of a joint configuration over both visible and hidden units depends on the energy of that joint configuration compared with the energy of all other joint configurations.
- The probability of a configuration of the visible units is the sum of the probabilities of all the joint configurations that contain it.

$$p(\mathbf{v}, \mathbf{h}) = \frac{e^{-E(\mathbf{v}, \mathbf{h})}}{\sum_{\mathbf{u}, \mathbf{g}} e^{-E(\mathbf{u}, \mathbf{g})}}$$

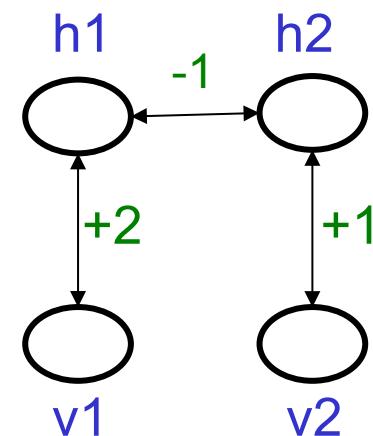
partition function

$$p(\mathbf{v}) = \frac{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}}{\sum_{\mathbf{u}, \mathbf{g}} e^{-E(\mathbf{u}, \mathbf{g})}}$$

v	h	-E	e^{-E}	$p(v, h)$	$p(v)$
1 1	1 1	2	7.39	.186	
1 1	1 0	2	7.39	.186	
1 1	0 1	1	2.72	.069	0.466
1 1	0 0	0	1	.025	
1 0	1 1	1	2.72	.069	
1 0	1 0	2	7.39	.186	
1 0	0 1	0	1	.025	0.305
1 0	0 0	0	1	.025	
0 1	1 1	0	1	.025	
0 1	1 0	0	1	.025	
0 1	0 1	1	2.72	.069	0.144
0 1	0 0	0	1	.025	
0 0	1 1	-1	0.37	.009	
0 0	1 0	0	1	.025	
0 0	0 1	0	1	.025	0.084
0 0	0 0	0	1	.025	

39.70

An example of how weights define a distribution



Getting a sample from the model

- If there are more than a few hidden units, we cannot compute the normalizing term (the partition function) because it has exponentially many terms.
- So we use Markov Chain Monte Carlo to get samples from the model starting from a random global configuration:
 - Keep picking units at random and allowing them to stochastically update their states based on their energy gaps.
- Run the Markov chain until it reaches its stationary distribution (thermal equilibrium at a temperature of 1).
 - The probability of a global configuration is then related to its energy by the Boltzmann distribution.
$$p(\mathbf{v}, \mathbf{h}) \propto e^{-E(\mathbf{v}, \mathbf{h})}$$

Getting a sample from the posterior distribution over hidden configurations for a given data vector

- The number of possible hidden configurations is exponential so we need MCMC to sample from the posterior.
 - It is just the same as getting a sample from the model, except that we keep the visible units clamped to the given data vector.
 - Only the hidden units are allowed to change states
- Samples from the posterior are required for learning the weights. Each hidden configuration is an “explanation” of an observed visible configuration. Better explanations have lower energy.

Neural Networks for Machine Learning

Lecture 12a

The Boltzmann Machine learning algorithm

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

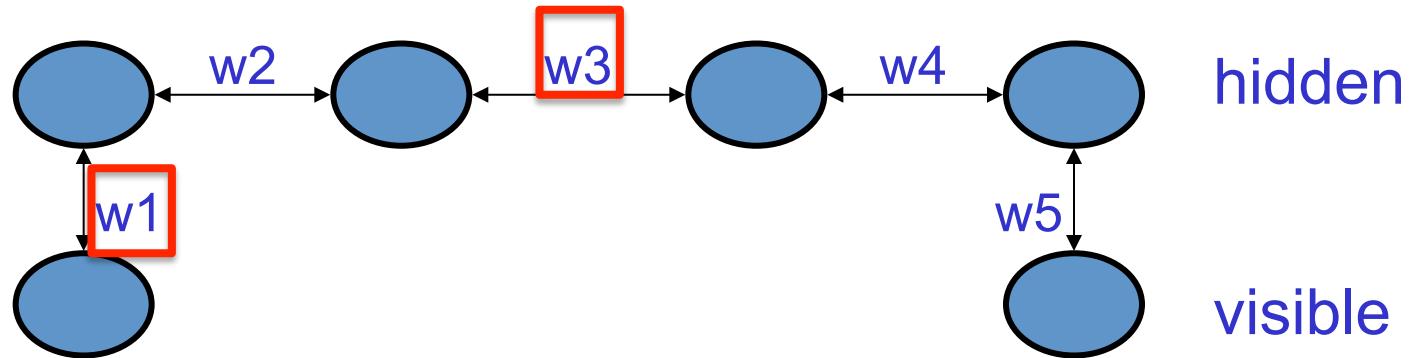
Abdel-rahman Mohamed

The goal of learning

- We want to maximize the product of the probabilities that the Boltzmann machine assigns to the binary vectors in the training set.
 - This is equivalent to maximizing the sum of the log probabilities that the Boltzmann machine assigns to the training vectors.
- It is also equivalent to maximizing the probability that we would obtain exactly the N training cases if we did the following
 - Let the network settle to its stationary distribution N different times with no external input.
 - Sample the visible vector once each time.

Why the learning could be difficult

Consider a chain of units with visible units at the ends



If the training set consists of $(1,0)$ and $(0,1)$ we want the product of all the weights to be negative.

So to know how to change w_1 or w_5 we must know w_3 .

A very surprising fact

- Everything that one weight needs to know about the other weights and the data is contained in the difference of two correlations.

$$\frac{\partial \log p(\mathbf{v})}{\partial w_{ij}} = \left\langle s_i s_j \right\rangle_{\mathbf{v}} - \left\langle s_i s_j \right\rangle_{model}$$

Derivative of log probability of one training vector, \mathbf{v} under the model.

Expected value of product of states at thermal equilibrium when \mathbf{v} is clamped on the visible units

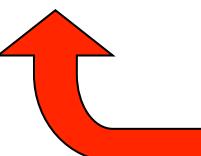
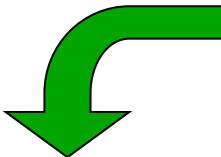
Expected value of product of states at thermal equilibrium with no clamping

$$\Delta w_{ij} \propto \left\langle s_i s_j \right\rangle_{data} - \left\langle s_i s_j \right\rangle_{model}$$

Why is the derivative so simple?

- The probability of a global configuration **at thermal equilibrium** is an exponential function of its energy.
 - So settling to equilibrium makes the log probability a linear function of the energy.
- The energy is a linear function of the weights and states, so:
$$-\frac{\partial E}{\partial w_{ij}} = s_i s_j$$
- The process of settling to thermal equilibrium propagates information about the weights.
 - We don't need backprop.

Why do we need the negative phase?

$$p(\mathbf{v}) = \frac{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}}{\sum_{\mathbf{u}} \sum_{\mathbf{g}} e^{-E(\mathbf{u}, \mathbf{g})}}$$


The positive phase finds hidden configurations that work well with \mathbf{v} and lowers their energies.

The negative phase finds the joint configurations that are the best competitors and raises their energies.

An inefficient way to collect the statistics required for learning

Hinton and Sejnowski (1983)

- **Positive phase:** Clamp a data vector on the visible units and set the hidden units to random binary states.
 - Update the hidden units one at a time until the network reaches thermal equilibrium at a temperature of 1.
 - Sample $\langle s_i s_j \rangle$ for every connected pair of units.
 - Repeat for all data vectors in the training set and average.
- **Negative phase:** Set **all** the units to random binary states.
 - Update **all** the units one at a time until the network reaches thermal equilibrium at a temperature of 1.
 - Sample $\langle s_i s_j \rangle$ for every connected pair of units.
 - Repeat many times (how many?) and average to get good estimates.

Neural Networks for Machine Learning

Lecture 12b

More efficient ways to get the statistics

ADVANCED MATERIAL: NOT ON QUIZZES OR FINAL TEST

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

A better way of collecting the statistics

- If we start from a random state, it may take a long time to reach thermal equilibrium.
 - Also, its very hard to tell when we get there.
 - Why not start from whatever state you ended up in last time you saw that datavector?
 - This stored state is called a “particle”.
- Using particles that persist to get a “warm start” has a big advantage:
- If we were at equilibrium last time and we only changed the weights a little, we should only need a few updates to get back to equilibrium.

Neal's method for collecting the statistics (Neal 1992)

- **Positive phase:** Keep a set of “data-specific particles”, one per training case. Each particle has a current value that is a configuration of the hidden units.
 - Sequentially update all the hidden units a few times in each particle with the relevant datavector clamped.
 - For every connected pair of units, average $s_i s_j$ over all the data-specific particles.
- **Negative phase:** Keep a set of “fantasy particles”. Each particle has a value that is a global configuration.
 - Sequentially update all the units in each fantasy particle a few times.
 - For every connected pair of units, average $s_i s_j$ over all the fantasy particles.

$$\Delta w_{ij} \propto \langle s_i s_j \rangle_{data} - \langle s_i s_j \rangle_{model}$$

Adapting Neal's approach to handle mini-batches

- Neal's approach does not work well with mini-batches.
 - By the time we get back to the same datavector again, the weights will have been updated many times.
 - But the data-specific particle will not have been updated so it may be far from equilibrium.
- A strong assumption about how we understand the world:
 - When a datavector is clamped, we will assume that the set of good explanations (i.e. hidden unit states) is uni-modal.
 - i.e. we restrict ourselves to learning models in which one sensory input vector does not have multiple very different explanations.

The simple mean field approximation

- If we want to get the statistics right, we need to update the units stochastically and sequentially.

$$\text{prob}(s_i = 1) = \sigma\left(b_i + \sum_j s_j w_{ij}\right)$$

- But if we are in a hurry we can use probabilities instead of binary states and update the units in parallel.

$$p_i^{t+1} = \sigma\left(b_i + \sum_j p_j^t w_{ij}\right)$$

- To avoid biphasic oscillations we can use damped mean field.

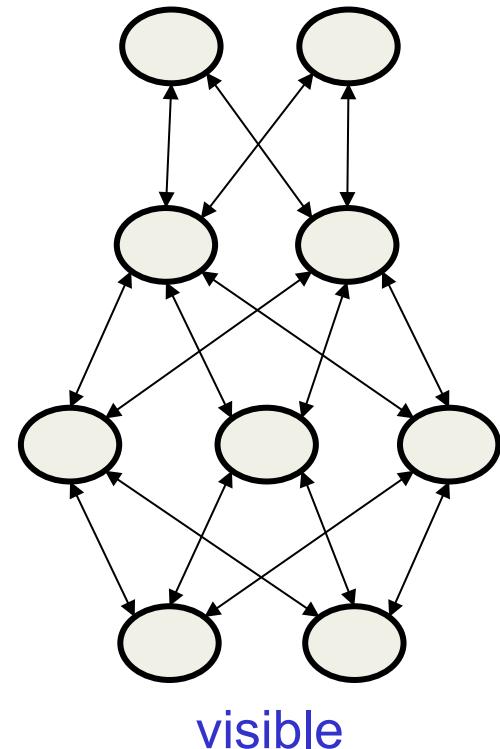
$$p_i^{t+1} = \lambda p_i^t + (1 - \lambda) \sigma\left(b_i + \sum_j p_j^t w_{ij}\right)$$

An efficient mini-batch learning procedure for Boltzmann Machines (Salakhutdinov & Hinton 2012)

- **Positive phase:** Initialize all the hidden probabilities at 0.5.
 - Clamp a datavector on the visible units.
 - Update all the hidden units in parallel until convergence using mean field updates.
 - After the net has converged, record $p_i p_j$ for every connected pair of units and average this over all data in the mini-batch.
- **Negative phase:** Keep a set of “fantasy particles”. Each particle has a value that is a global configuration.
 - Sequentially update all the units in each fantasy particle a few times.
 - For every connected pair of units, average $s_i s_j$ over all the fantasy particles.

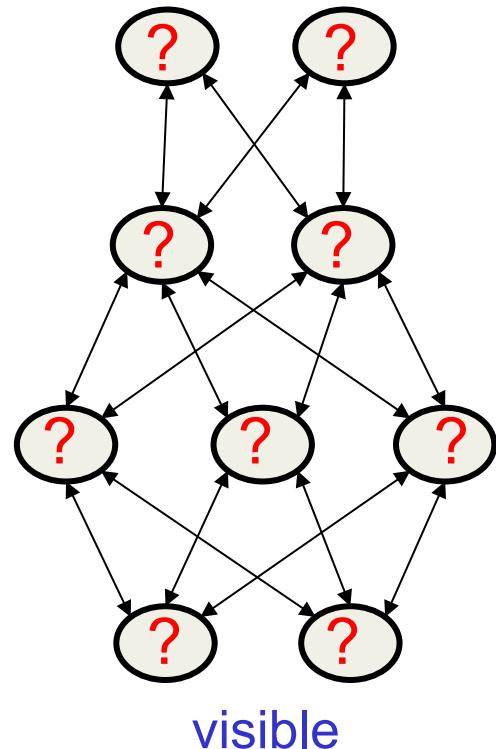
Making the updates more parallel

- In a general Boltzmann machine, the stochastic updates of units need to be sequential.
- There is a special architecture that allows alternating parallel updates which are much more efficient:
 - No connections within a layer.
 - No skip-layer connections.
- This is called a Deep Boltzmann Machine (DBM)
 - It's a general Boltzmann machine with a lot of missing connections.



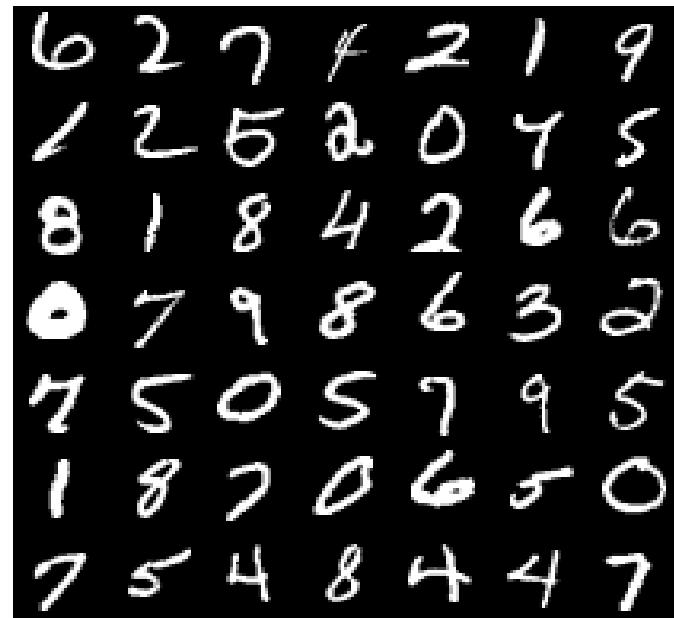
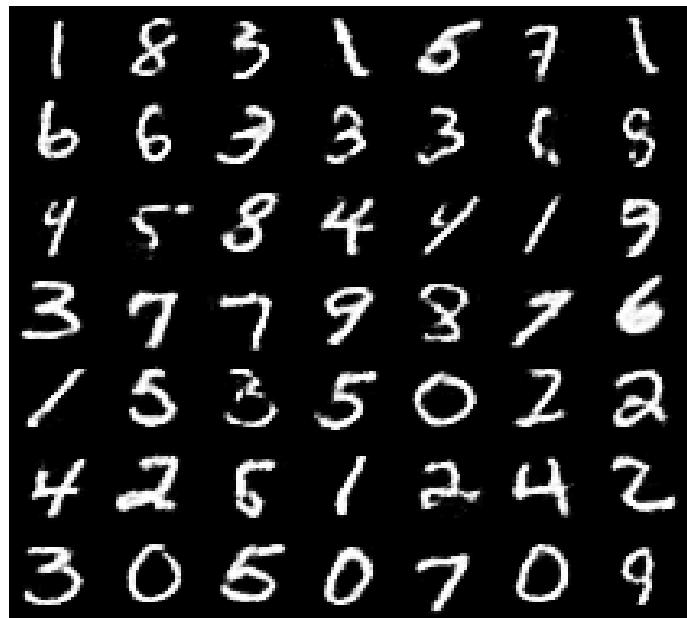
Making the updates more parallel

- In a general Boltzmann machine, the stochastic updates of units need to be sequential.
- There is a special architecture that allows alternating parallel updates which are much more efficient:
 - No connections within a layer.
 - No skip-layer connections.
- This is called a Deep Boltzmann Machine (DBM)
 - It's a general Boltzmann machine with a lot of missing connections.



Can a DBM learn a good model of the MNIST digits?

Do samples from the model look like real data?



A puzzle

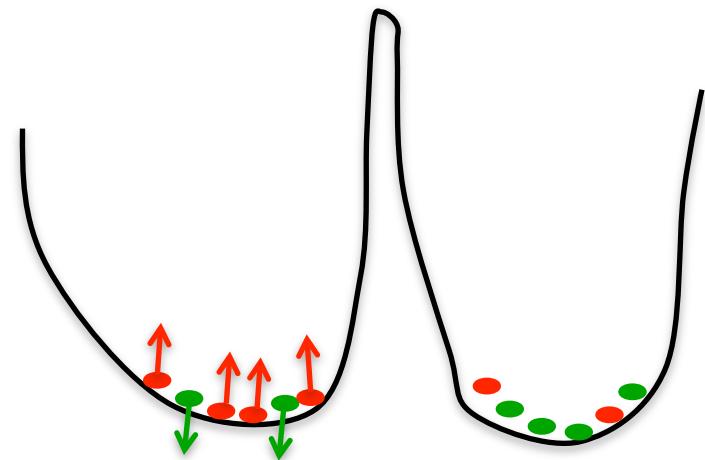
- Why can we estimate the “negative phase statistics” well with only 100 negative examples to characterize the whole space of possible configurations?
 - For all interesting problems the GLOBAL configuration space is highly multi-modal.
 - How does it manage to find and represent all the modes with only 100 particles?

The learning raises the effective mixing rate.

- The learning interacts with the Markov chain that is being used to gather the “negative statistics” (*i.e.* the data-independent statistics).
 - We cannot analyse the learning by viewing it as an outer loop and the gathering of statistics as an inner loop.
- Wherever the fantasy particles outnumber the positive data, the energy surface is raised.
 - This makes the fantasies rush around hyperactively.
 - They move around MUCH faster than the mixing rate of the Markov chain defined by the static current weights.

How fantasy particles move between the model's modes

- If a mode has more fantasy particles than data, the energy surface is raised until the fantasy particles escape.
 - This can overcome energy barriers that would be too high for the Markov chain to jump in a reasonable time.
- The energy surface is being changed to help **mixing** in addition to defining the model.
- Once the fantasy particles have filled in a hole, they rush off somewhere else to deal with the next problem.
 - They are like investigative journalists.



This minimum will get filled in by the learning until the fantasy particles escape.

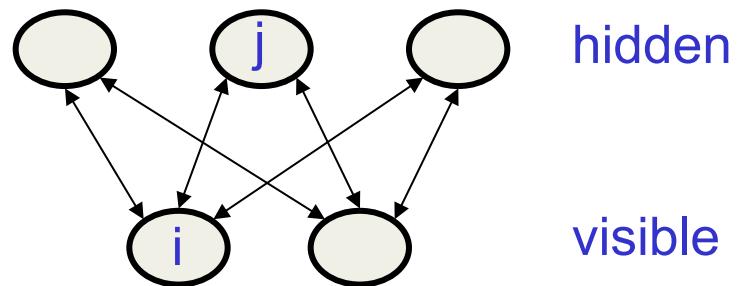
Neural Networks for Machine Learning

Lecture 12c Restricted Boltzmann Machines

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

Restricted Boltzmann Machines

- We restrict the connectivity to make inference and learning easier.
 - Only one layer of hidden units.
 - No connections between hidden units.
- In an RBM it only takes one step to reach thermal equilibrium when the visible units are clamped.
 - So we can quickly get the exact value of : $\langle v_i h_j \rangle_v$

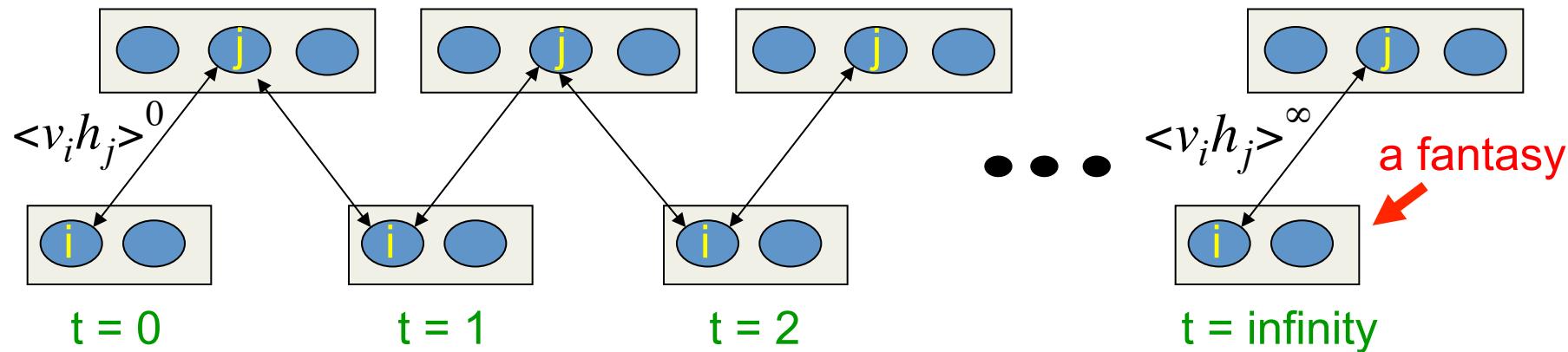


$$p(h_j = 1) = \frac{1}{1 + e^{-\left(b_j + \sum_{i \in vis} v_i w_{ij}\right)}}$$

PCD: An efficient mini-batch learning procedure for Restricted Boltzmann Machines (Tieleman, 2008)

- Positive phase: Clamp a datavector on the visible units.
 - Compute the exact value of $\langle v_i h_j \rangle$ for all pairs of a visible and a hidden unit.
 - For every connected pair of units, average $\langle v_i h_j \rangle$ over all data in the mini-batch.
- Negative phase: Keep a set of “fantasy particles”. Each particle has a value that is a global configuration.
 - Update each fantasy particle a few times using alternating parallel updates.
 - For every connected pair of units, average $v_i h_j$ over all the fantasy particles.

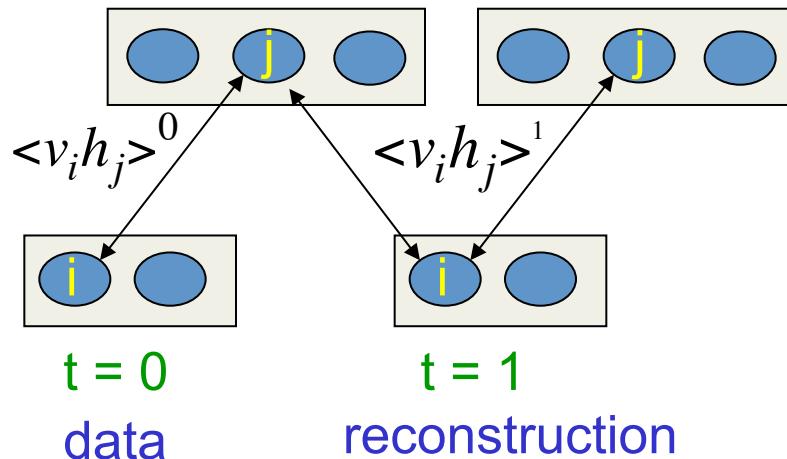
A picture of an inefficient version of the Boltzmann machine learning algorithm for an RBM



Start with a training vector on the visible units. Then alternate between updating all the hidden units in parallel and updating all the visible units in parallel.

$$\Delta w_{ij} = \varepsilon (\langle v_i h_j \rangle^0 - \langle v_i h_j \rangle^\infty)$$

Contrastive divergence: A very surprising short-cut



$$\Delta w_{ij} = \varepsilon (\langle v_i h_j \rangle^0 - \langle v_i h_j \rangle^1)$$

Start with a training vector on the visible units.

Update all the hidden units in parallel.

Update the all the visible units in parallel to get a “reconstruction”.

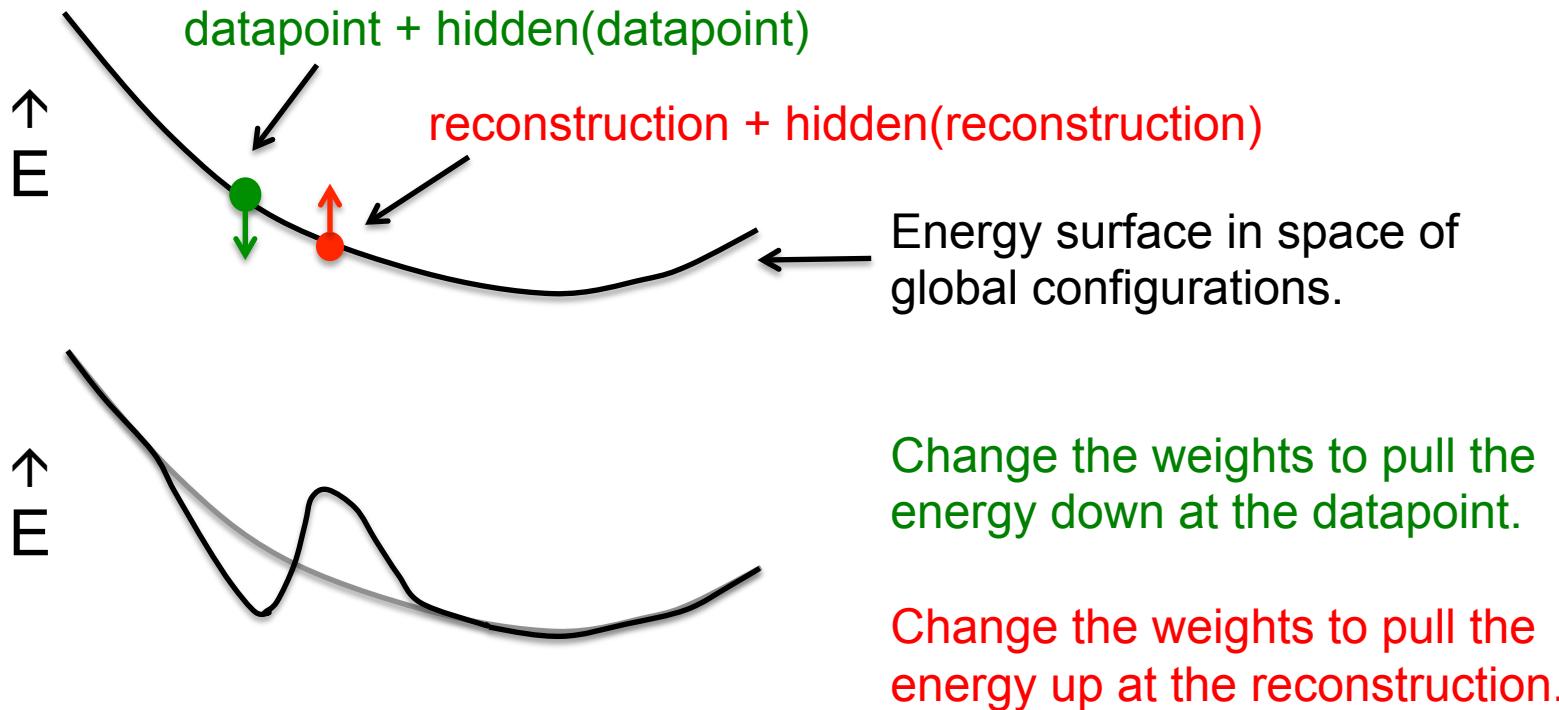
Update the hidden units again.

This is not following the gradient of the log likelihood. But it works well.

Why does the shortcut work?

- If we start at the data, the Markov chain wanders away from the data and towards things that it likes more.
 - We can see what direction it is wandering in after only a few steps.
 - When we know the weights are bad, it is a waste of time to let it go all the way to equilibrium.
- All we need to do is lower the probability of the confabulations it produces after one full step and raise the probability of the data.
 - Then it will stop wandering away.
 - The learning cancels out once the confabulations and the data have the same distribution.

A picture of contrastive divergence learning



When does the shortcut fail?

- We need to worry about regions of the data-space that the model likes but which are very far from any data.
 - These low energy holes cause the normalization term to be big and we cannot sense them if we use the shortcut.
 - Persistent particles would eventually fall into a hole, cause it to fill up then move on to another hole.
- A good compromise between speed and correctness is to start with small weights and use CD1 (*i.e.* use one full step to get the “negative data”).
 - Once the weights grow, the Markov chain mixes more slowly so we use CD3.
 - Once the weights have grown more we use CD10.

Neural Networks for Machine Learning

Lecture 12d

An example of Contrastive Divergence Learning

Geoffrey Hinton

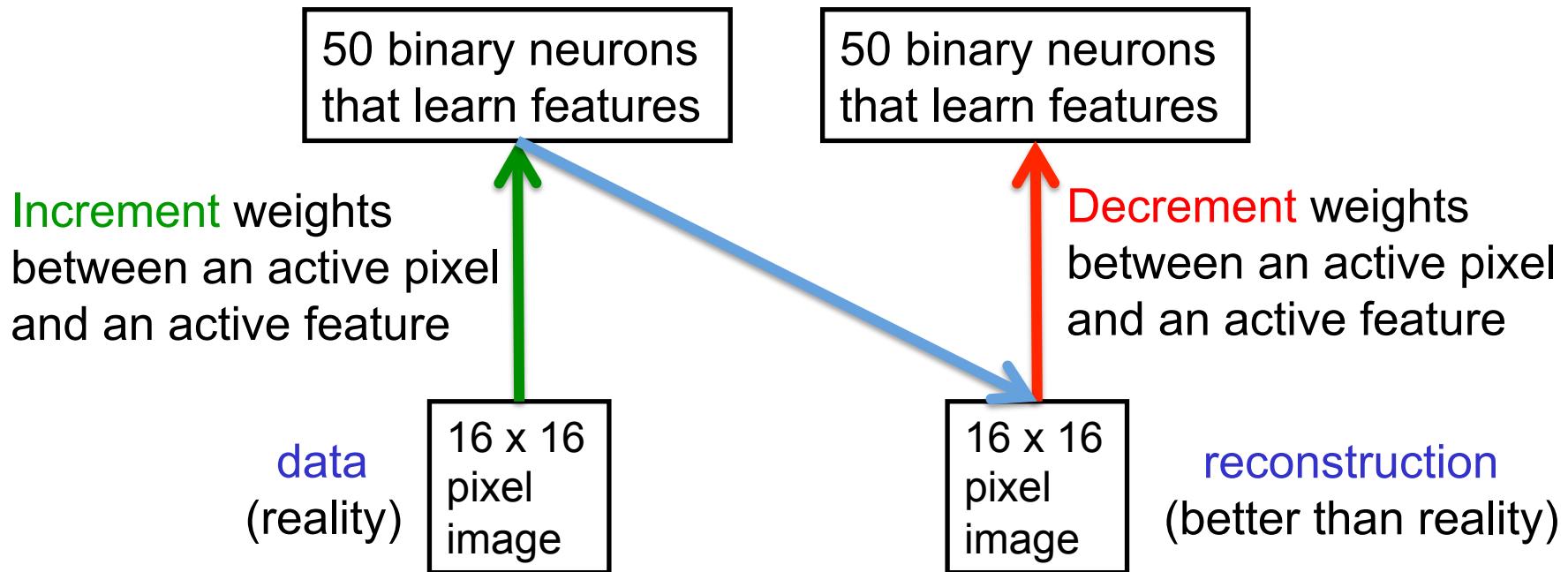
Nitish Srivastava,

Kevin Swersky

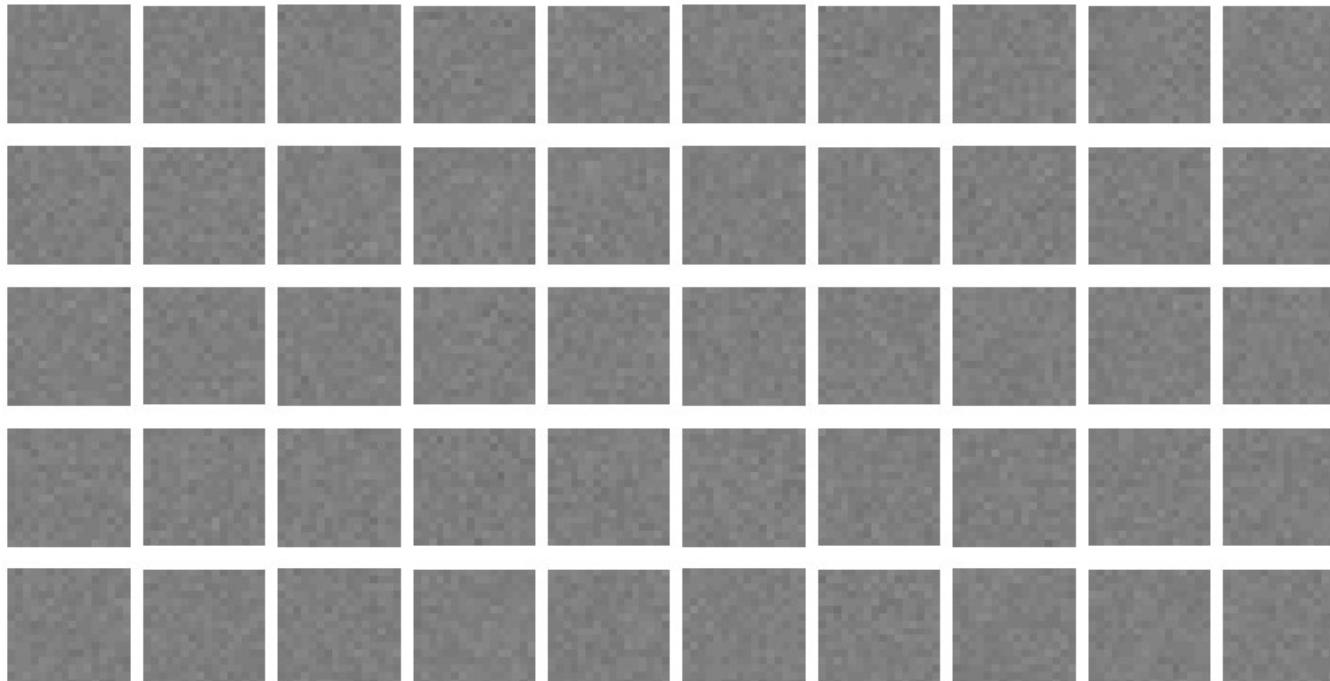
Tijmen Tieleman

Abdel-rahman Mohamed

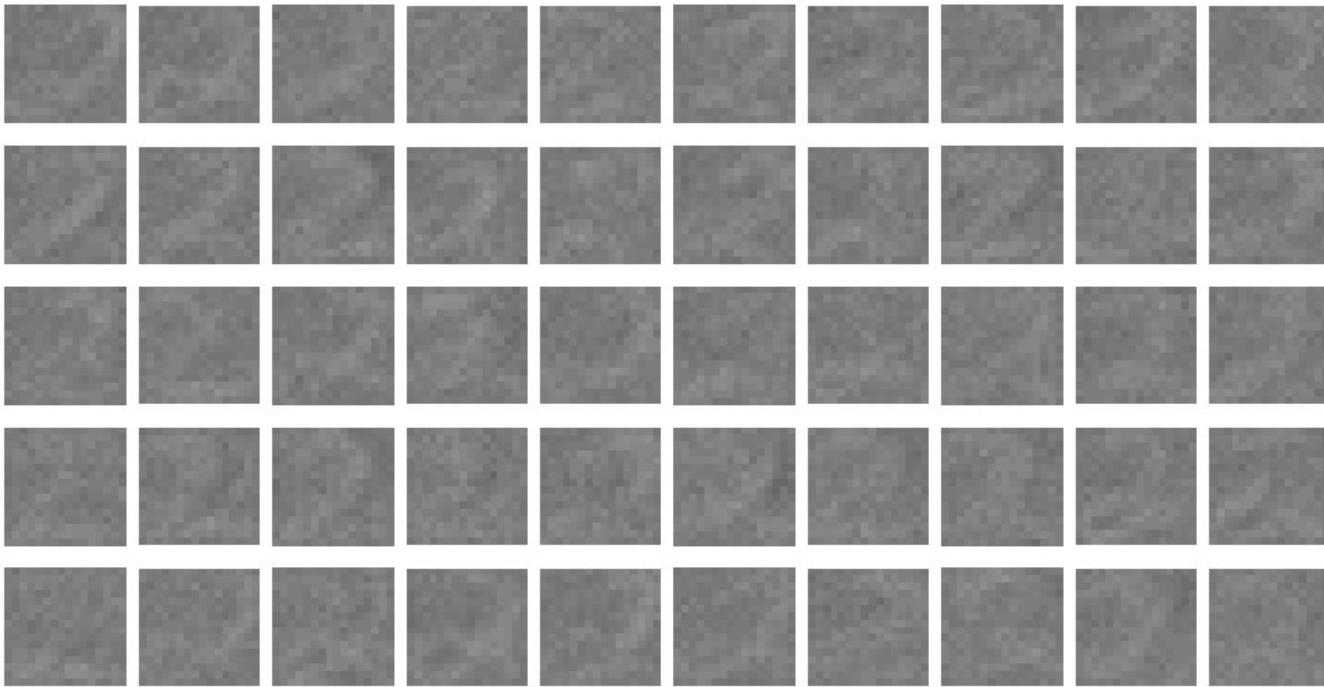
How to learn a set of features that are good for reconstructing images of the digit 2

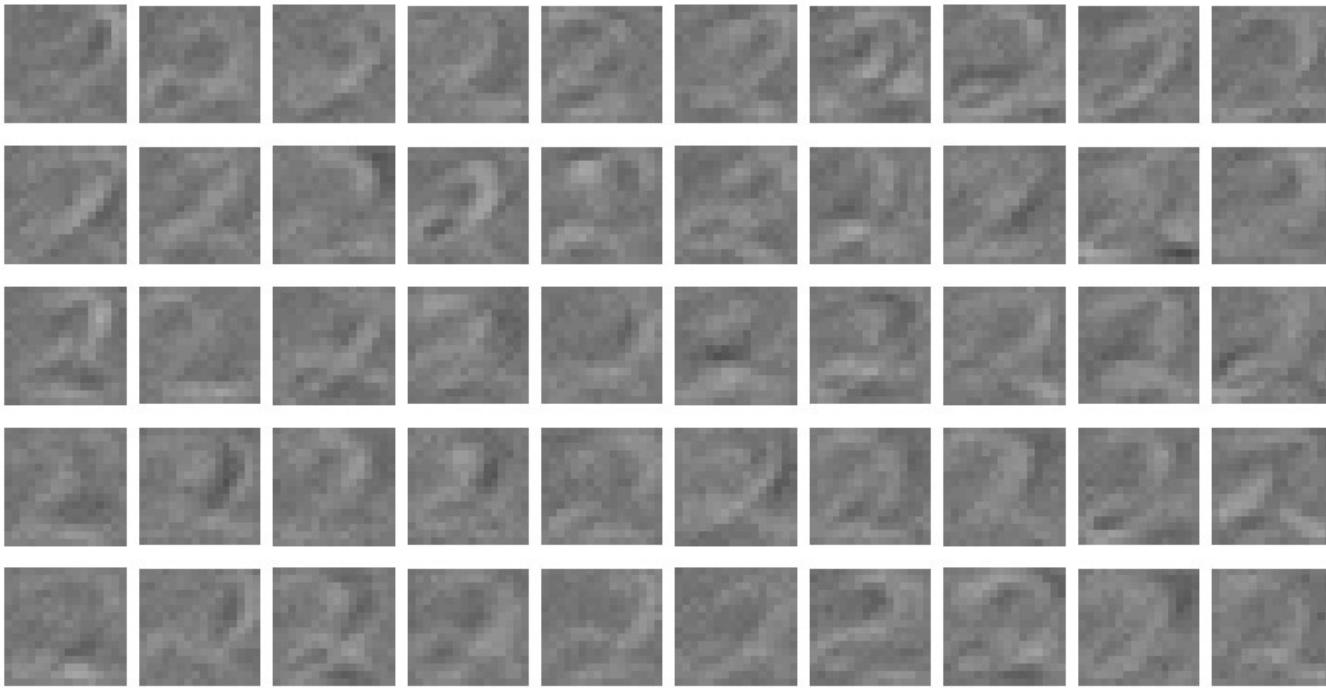


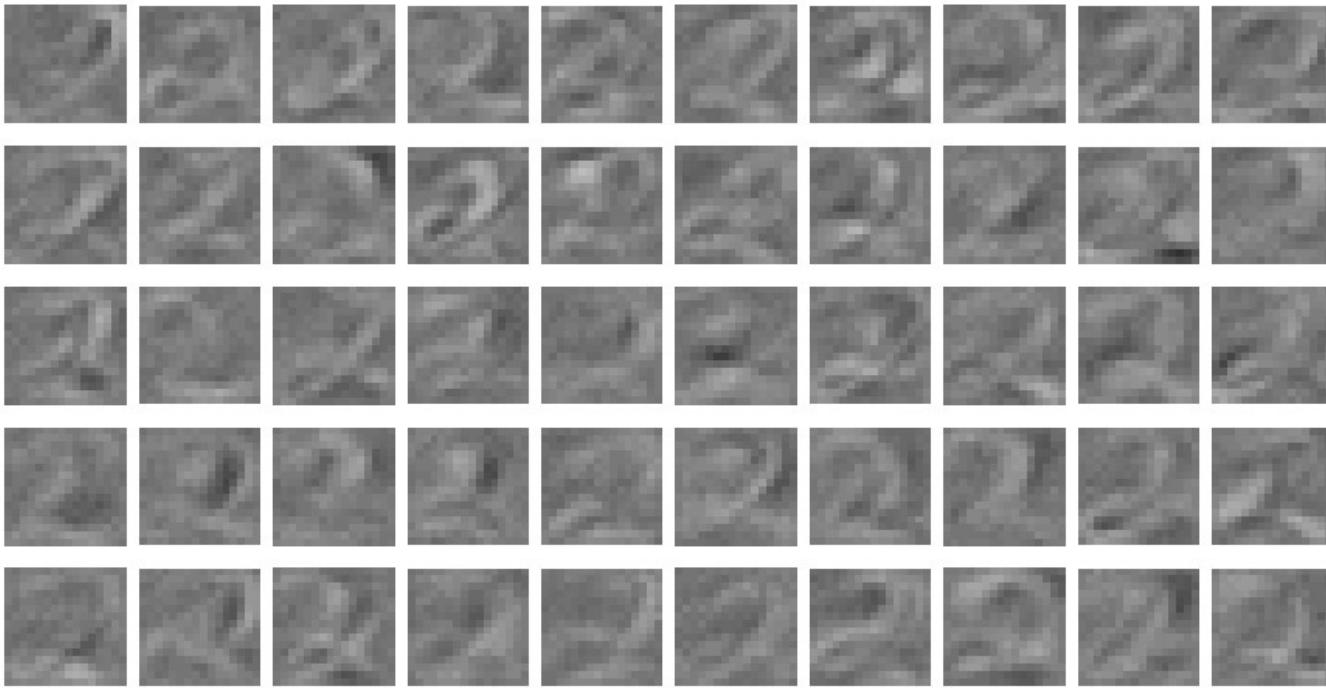
The weights of the 50 feature detectors

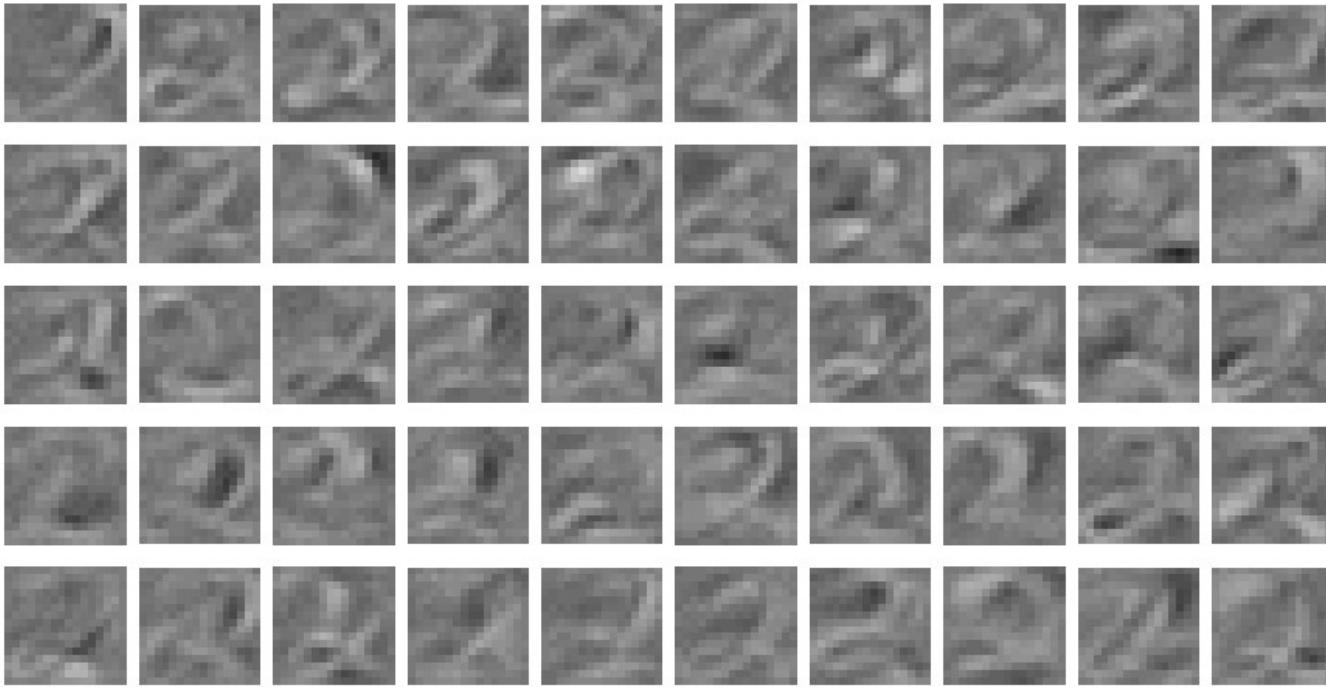


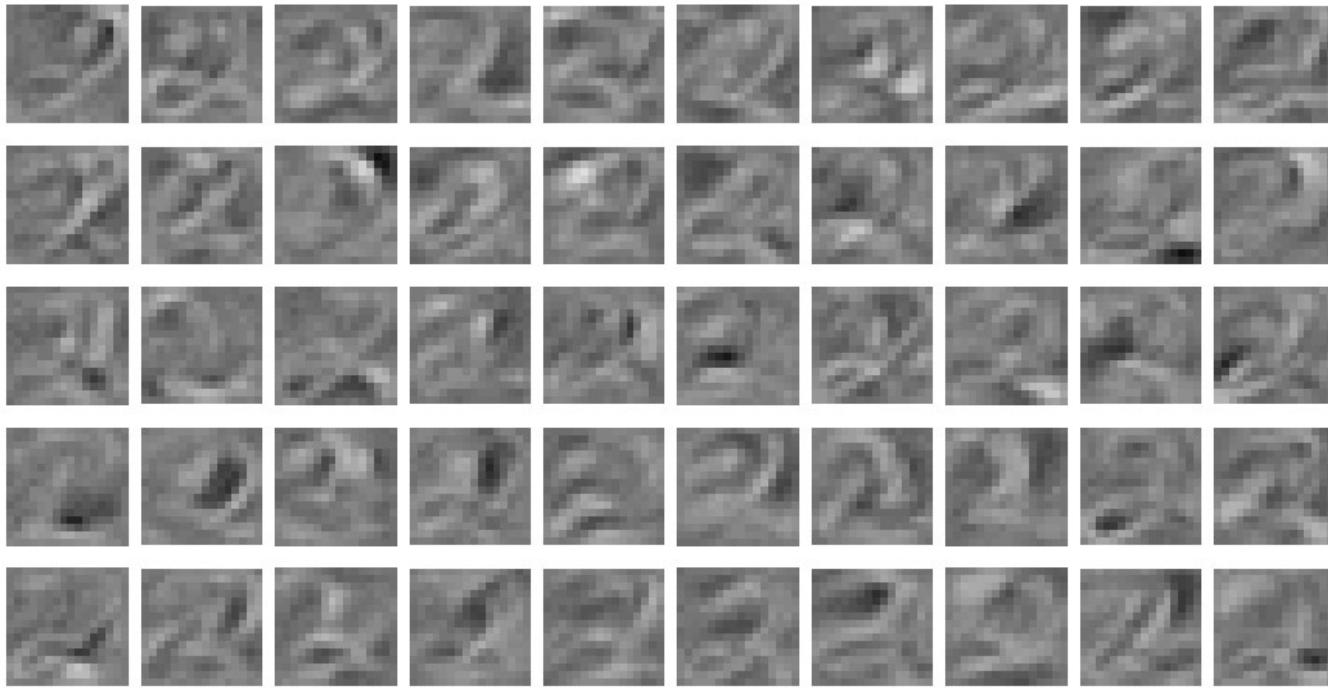
We start with small random weights to break symmetry

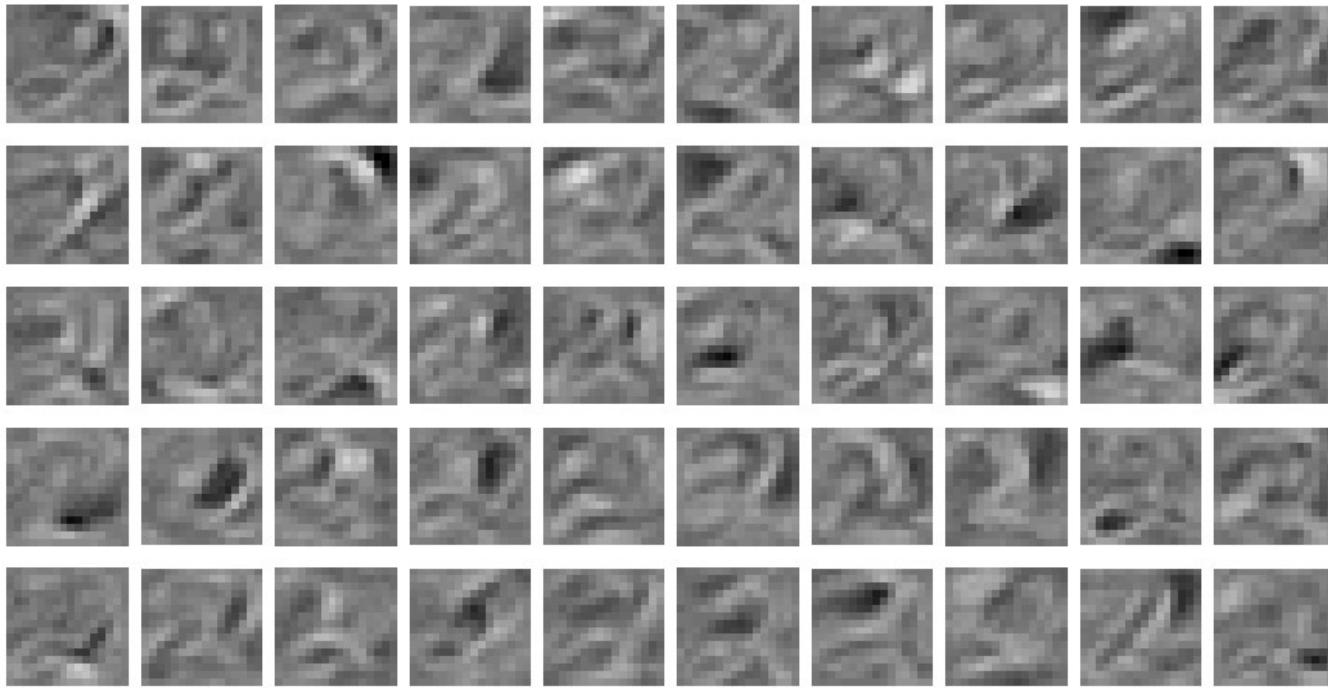


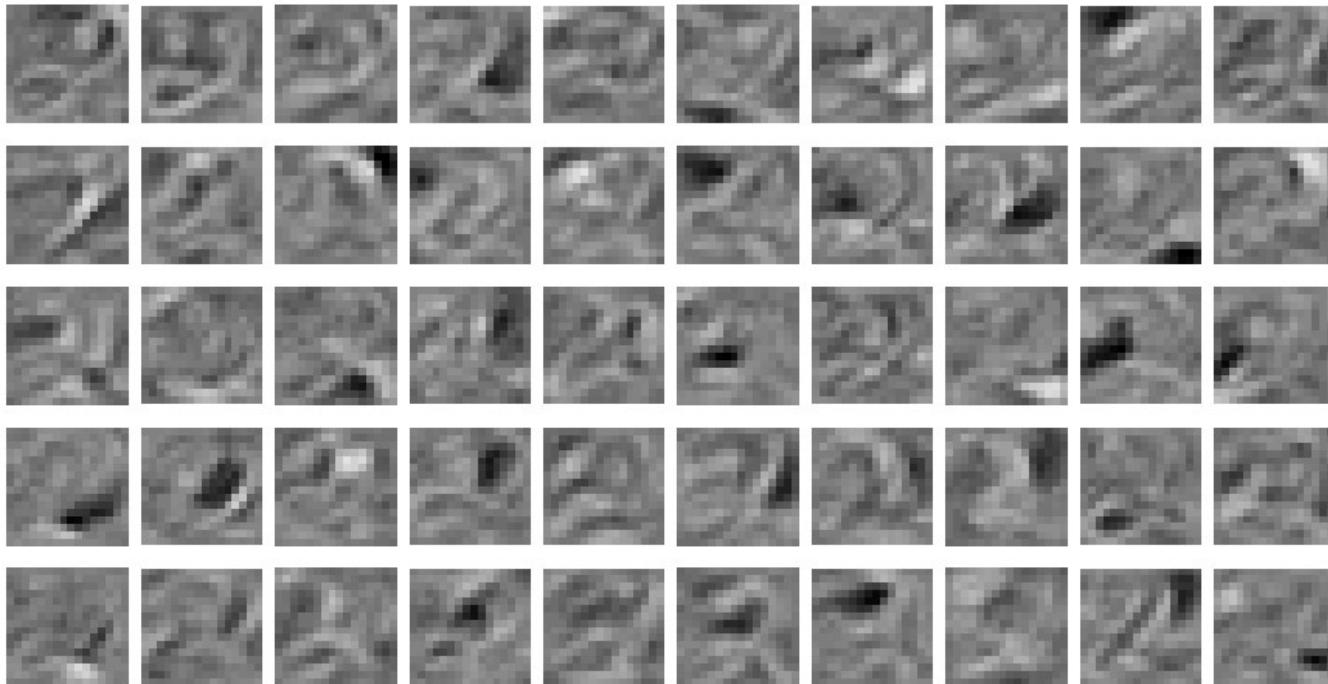


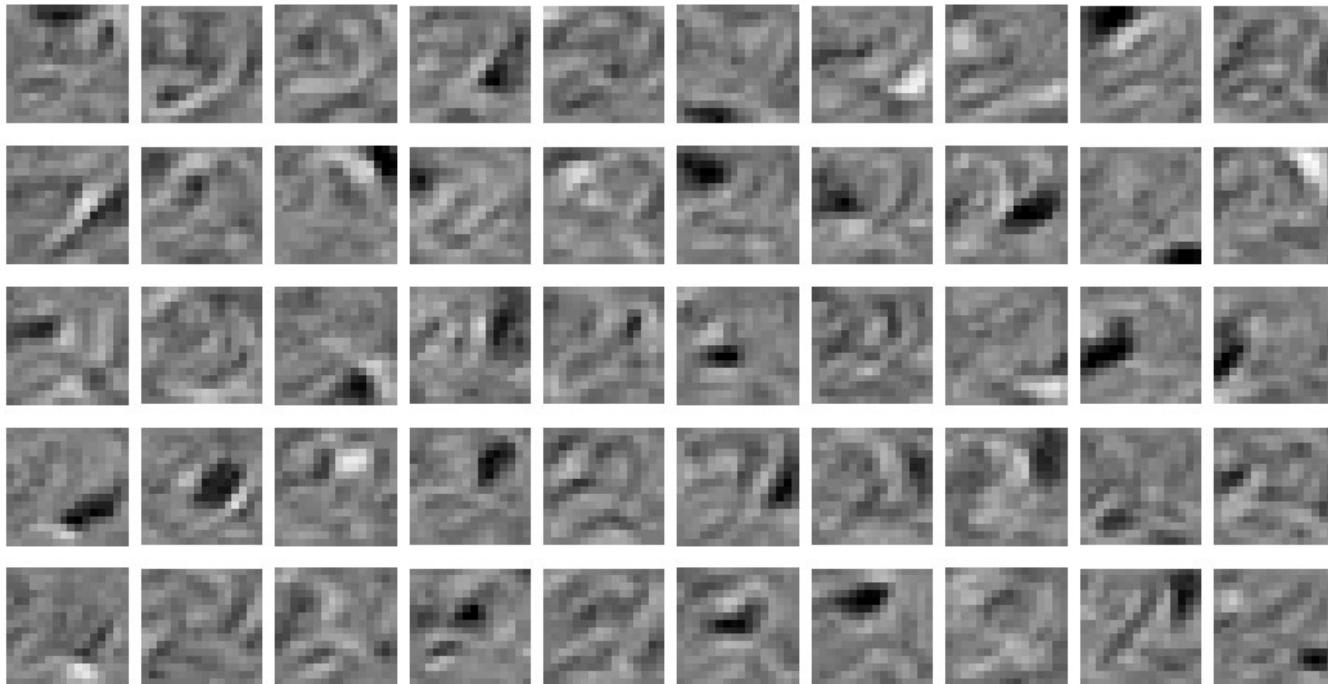




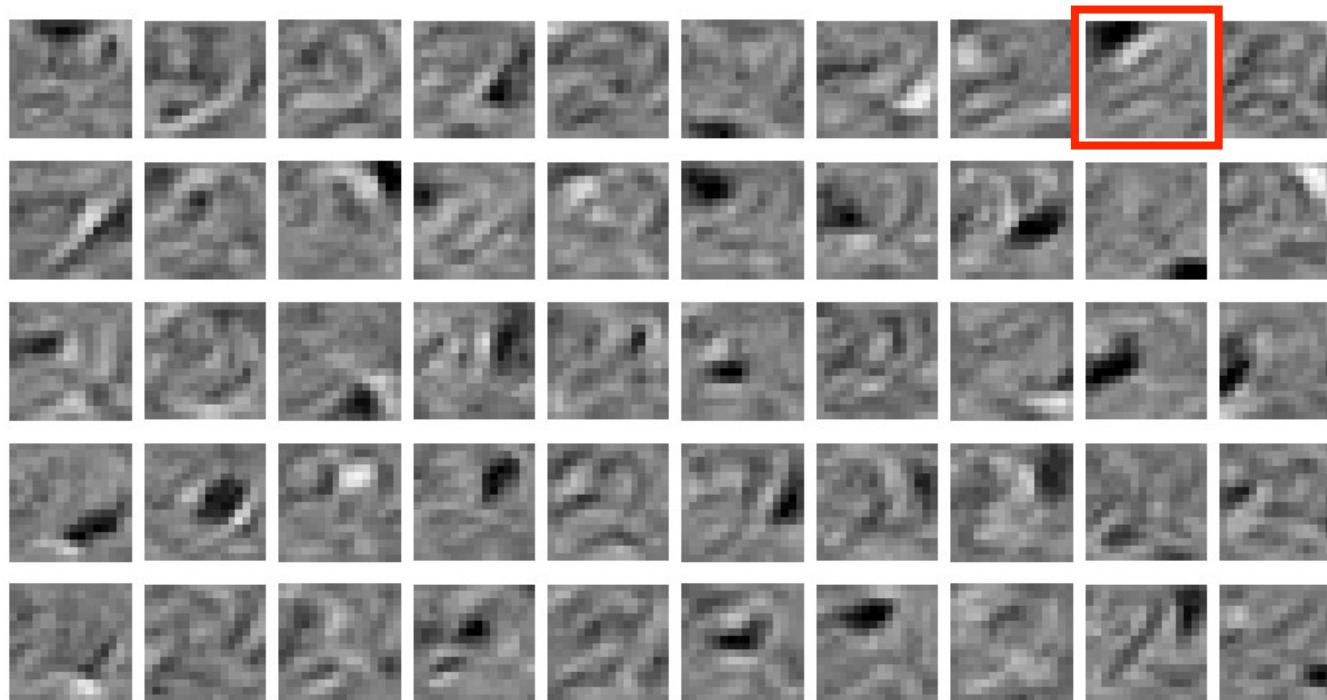




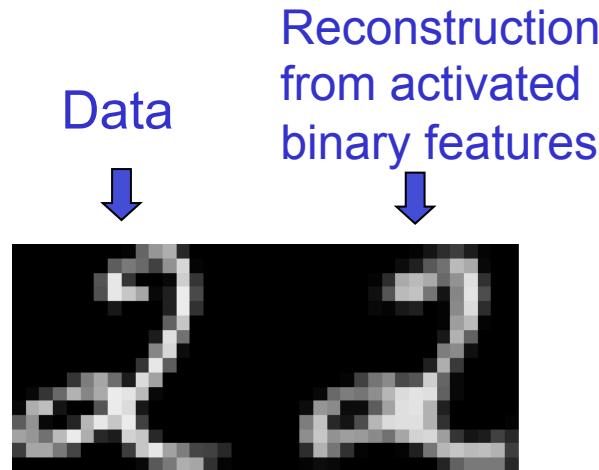




The final 50×256 weights: Each neuron grabs a different feature



How well can we reconstruct digit images from the binary feature activations?



New test image from the digit class that the model was trained on

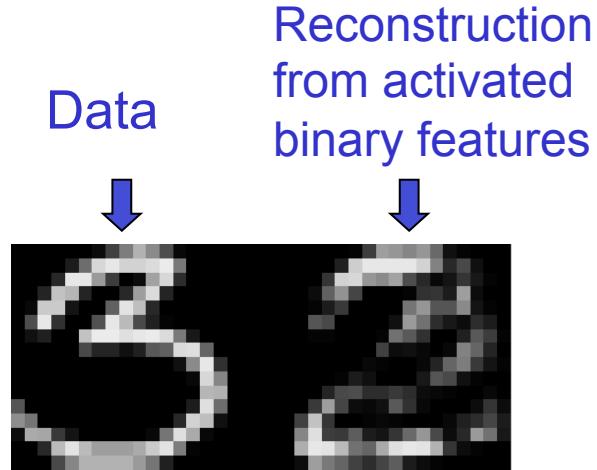
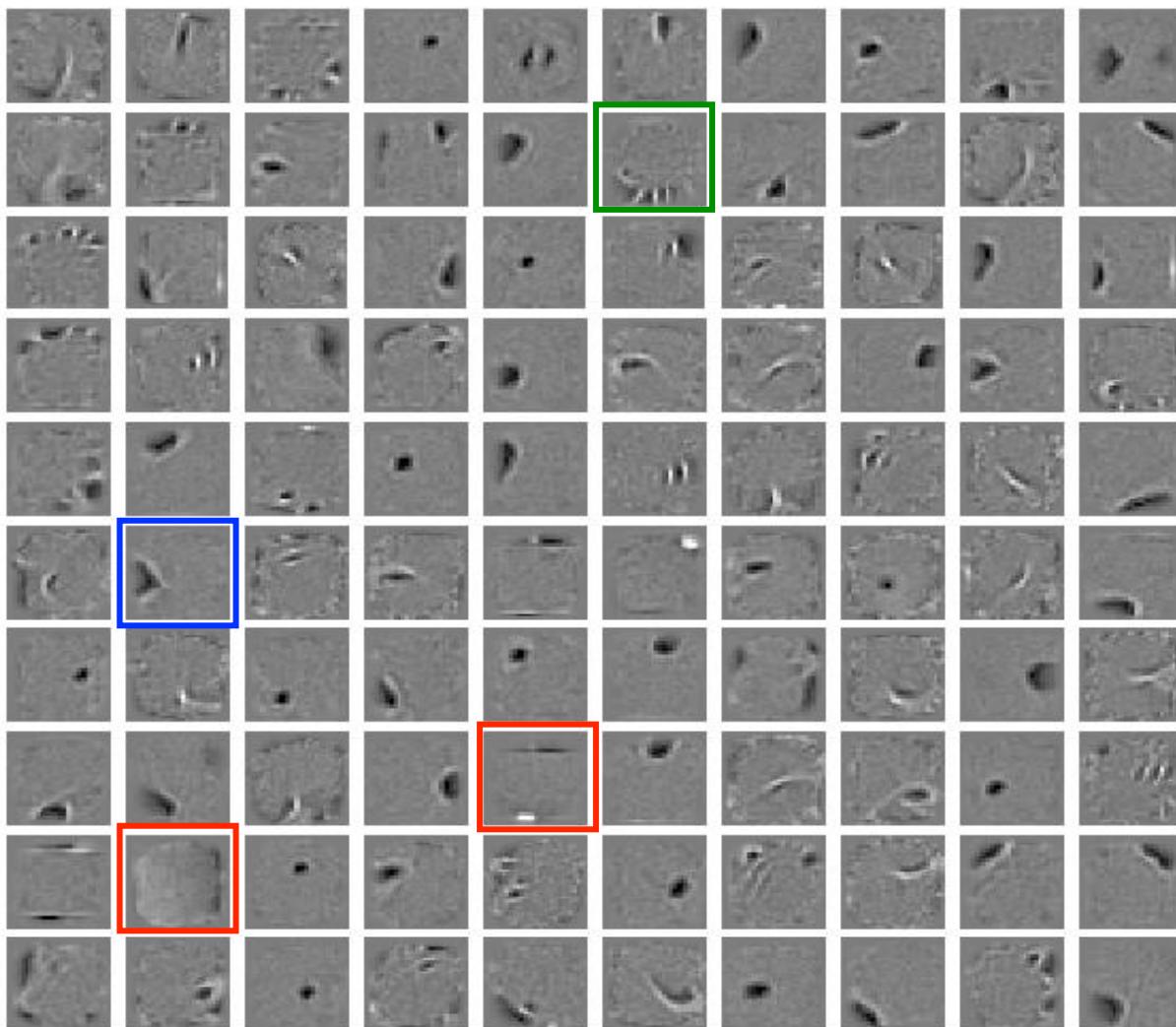


Image from an unfamiliar digit class

The network tries to see every image as a 2.



Some features learned in the first hidden layer of a model of all 10 digit classes using 500 hidden units.

Neural Networks for Machine Learning

Lecture 12e RBMs for collaborative filtering

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

Collaborative filtering: The Netflix competition

- You are given most of the ratings that half a million Users gave to 18,000 Movies on a scale from 1 to 5.
 - Each user only rates a small fraction of the movies.
- You have to predict the ratings users gave to the held out movies.
 - If you win you get \$1000,000

	M1	M2	M3	M4	M5	M6
U1				3		
U2	5		1			
U3		3	5			
U4	4		?			5
U5			4			
U6					2	

Lets use a “language model”

The data is strings of triples of the form: User, Movie, rating.

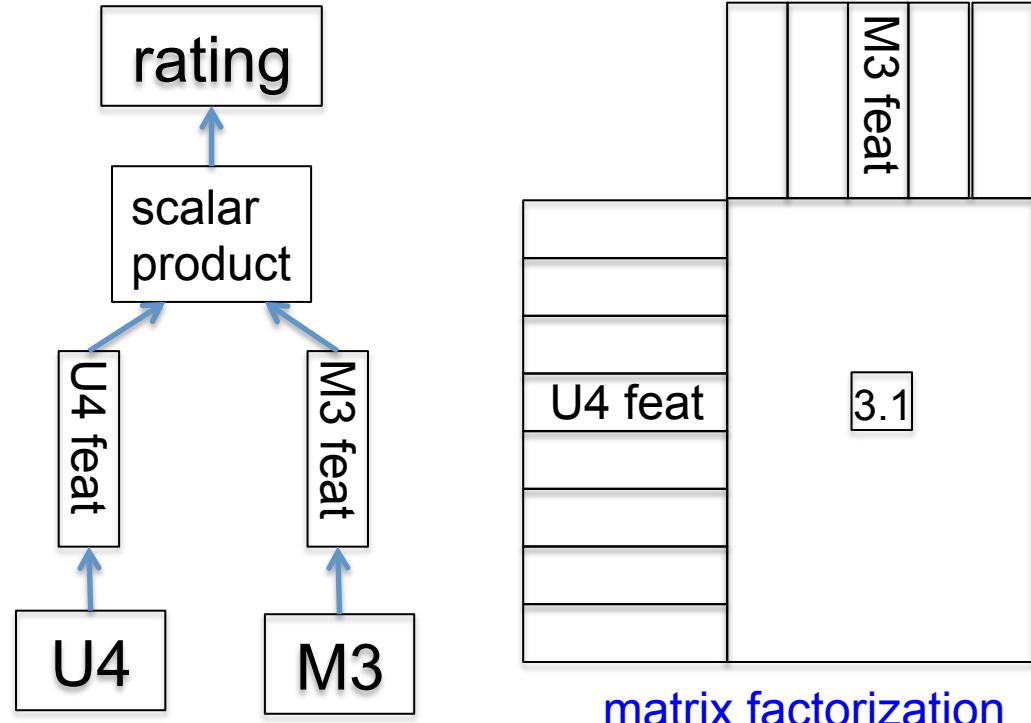
U2 M1 5

U2 M3 1

U4 M1 4

U4 M3 ?

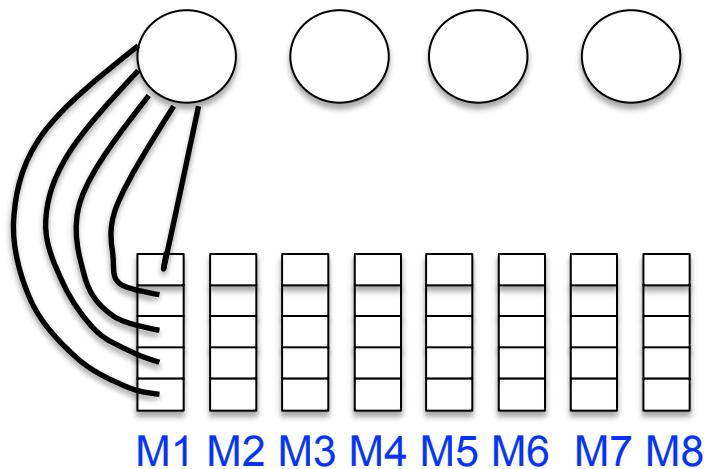
All we have to do is to predict the next “word” well and we will get rich.



An RBM alternative to matrix factorization

- Suppose we treat each user as a training case.
 - A user is a vector of movie ratings.
 - There is one visible unit per movie and its a 5-way softmax.
 - The CD learning rule for a softmax is the same as for a binary unit.
 - There are ~100 hidden units.
- One of the visible values is unknown.
 - It needs to be filled in by the model.

about 100 binary hidden units



How to avoid dealing with all those missing ratings

- For each user, use an RBM that only has visible units for the movies the user rated.
- So instead of one RBM for all users, we have a different RBM for every user.
 - All these RBMs use the same hidden units.
 - The weights from each hidden unit to each movie are shared by all the users who rated that movie.
- Each user-specific RBM only gets one training case!
 - But the weight-sharing makes this OK.
- The models are trained with CD1 then CD3, CD5 & CD9.

How well does it work?(Salakhutdinov *et al.* 2007)

- RBMs work about as well as matrix factorization methods, but they give very different errors.
 - So averaging the predictions of RBMs with the predictions of matrix-factorization is a big win.
- The winning group used multiple different RBM models in their average of over a hundred models.
 - Their main models were matrix factorization and RBMs (I think).

Neural Networks for Machine Learning

Lecture 13a

The ups and downs of backpropagation

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

A brief history of backpropagation

- The backpropagation algorithm for learning multiple layers of features was invented several times in the 70's and 80's:
 - Bryson & Ho (1969) linear
 - Werbos (1974)
 - Rumelhart et. al. in 1981
 - Parker (1985)
 - LeCun (1985)
 - Rumelhart et. al. (1985)
- Backpropagation clearly had great promise for learning multiple layers of non-linear feature detectors.
- But by the late 1990's most serious researchers in machine learning had given up on it.
 - It was still widely used in psychological models and in practical applications such as credit card fraud detection.

Why backpropagation failed

- The popular explanation of why backpropagation failed in the 90's:
 - It could not make good use of multiple hidden layers.
(except in convolutional nets)
 - It did not work well in recurrent networks or deep auto-encoders.
 - Support Vector Machines worked better, required less expertise, produced repeatable results, and had much fancier theory.
- The real reasons it failed:
 - Computers were thousands of times too slow.
 - Labeled datasets were hundreds of times too small.
 - Deep networks were too small and not initialized sensibly.
- These issues prevented it from being successful for tasks where it would eventually be a big win.

A spectrum of machine learning tasks

Typical Statistics-----Artificial Intelligence

- Low-dimensional data (e.g. less than 100 dimensions)
- Lots of noise in the data.
- Not much structure in the data.
The structure can be captured by a fairly simple model.
- The main problem is separating true structure from noise.
 - Not ideal for non-Bayesian neural nets. Try SVM or GP.
- High-dimensional data (e.g. more than 100 dimensions)
- The noise is not the main problem.
- There is a huge amount of structure in the data, but its too complicated to be represented by a simple model.
- The main problem is figuring out a way to represent the complicated structure so that it can be learned.
 - Let backpropagation figure it out.

Why Support Vector Machines were never a good bet for Artificial Intelligence tasks that need good representations

- View 1: SVM's are just a clever reincarnation of Perceptrons.
 - They expand the input to a (very large) layer of non-linear **non-adaptive** features.
 - They only have one layer of adaptive weights.
 - They have a very efficient way of fitting the weights that controls overfitting.
- View 2: SVM's are just a clever reincarnation of Perceptrons.
 - They use each input vector in the training set to define a **non-adaptive** “pheature”.
 - The global match between a test input and that training input.
 - They have a clever way of simultaneously doing feature selection and finding weights on the remaining features.

Historical document from AT&T Adaptive Systems Research Dept., Bell Labs

1. Jackel bets (one fancy dinner) that by March 14, 2000, people will understand quantitatively why big neural nets working on large databases are not so bad. (Understanding means that there will be clear conditions and bounds)

Vapnik bets (one fancy dinner) that Jackel is wrong.

But .. If Vapnik figures out the bounds and conditions, Vapnik still wins the bet.

2. Vapnik bets (one fancy dinner) that by March 14, 2005, no one in his right mind will use neural nets that are essentially like those used in 1995.

Jackel bets (one fancy dinner) that Vapnik is wrong

Neural Networks for Machine Learning

Lecture 13b Belief Nets

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

What is wrong with back-propagation?

- It requires labeled training data.
 - Almost all data is unlabeled.
- The learning time does not scale well
 - It is very slow in networks with multiple hidden layers.
 - Why?
- It can get stuck in poor local optima.
 - These are often quite good, but for deep nets they are far from optimal.
 - Should we retreat to models that allow convex optimization?

Overcoming the limitations of back-propagation by using unsupervised learning

- Keep the efficiency and simplicity of using a gradient method for adjusting the weights, but use it for modeling the structure of the sensory input.
 - Adjust the weights to maximize the probability that a generative model would have generated the sensory input.
 - If you want to do computer vision, first learn computer graphics.
- The learning objective for a generative model:
 - Maximise $p(x)$ not $p(y | x)$
- What kind of generative model should we learn?
 - An energy-based model like a Boltzmann machine?
 - A causal model made of idealized neurons?
 - A hybrid of the two?

Artificial Intelligence and Probability

“Many ancient Greeks supported Socrates opinion that deep, inexplicable thoughts came from the gods. Today’s equivalent to those gods is the erratic, even probabilistic neuron. It is more likely that increased randomness of neural behavior is the problem of the epileptic and the drunk, not the advantage of the brilliant.”

P.H. Winston, “Artificial Intelligence”, 1977. (The first AI textbook)

“All of this will lead to theories of computation which are much less rigidly of an all-or-none nature than past and present formal logic ... There are numerous indications to make us believe that this new system of formal logic will move closer to another discipline which has been little linked in the past with logic. This is thermodynamics primarily in the form it was received from Boltzmann.”

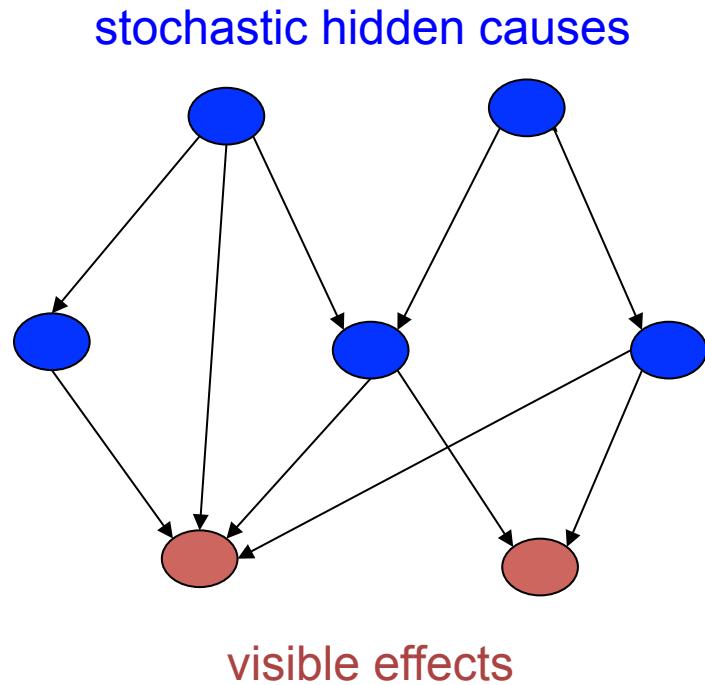
John von Neumann, “The Computer and the Brain”, 1958 (unfinished manuscript)

The marriage of graph theory and probability theory

- In the 1980's there was a lot of work in AI that used bags of rules for tasks such as medical diagnosis and exploration for minerals.
 - For practical problems, they had to deal with uncertainty.
 - They made up ways of doing this that did not involve probabilities!
- **Graphical models:** Pearl, Heckerman, Lauritzen, and many others showed that probabilities worked better.
 - Graphs were good for representing what depended on what.
 - Probabilities then had to be computed for nodes of the graph, given the states of other nodes.
- **Belief Nets:** For sparsely connected, directed acyclic graphs, clever inference algorithms were discovered.

Belief Nets

- A belief net is a directed acyclic graph composed of stochastic variables.
- We get to observe some of the variables and we would like to solve two problems:
- **The inference problem:** Infer the states of the unobserved variables.
- **The learning problem:** Adjust the interactions between variables to make the network more likely to generate the training data.

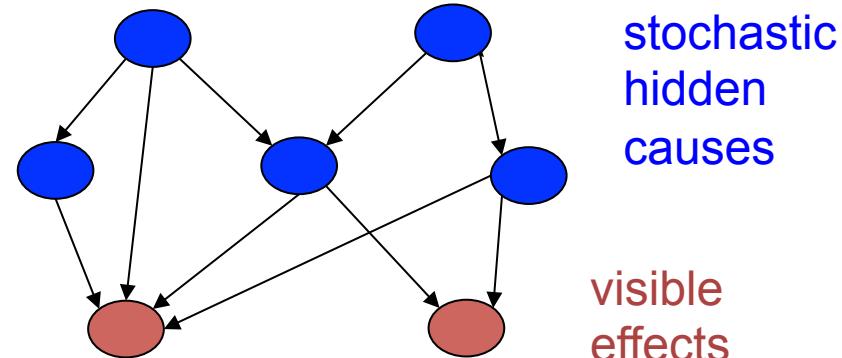


Graphical Models versus Neural Networks

- Early graphical models used experts to define the graph structure and the conditional probabilities.
 - The graphs were sparsely connected.
 - Researchers initially focused on doing correct inference, not on learning.
- For neural nets, learning was central. Hand-wiring the knowledge was not cool (OK, maybe a little bit).
 - Knowledge came from learning the training data.
- Neural networks did not aim for interpretability or sparse connectivity to make inference easy.
 - Nevertheless, there are neural network versions of belief nets.

Two types of generative neural network composed of stochastic binary neurons

- Energy-based: We connect binary stochastic neurons using **symmetric** connections to get a Boltzmann Machine.
 - If we restrict the connectivity in a special way, it is easy to learn a Boltzmann machine.
 - But then we only have one hidden layer.
- Causal: We connect binary stochastic neurons in a **directed acyclic graph** to get a Sigmoid Belief Net (Neal 1992).



Neural Networks for Machine Learning

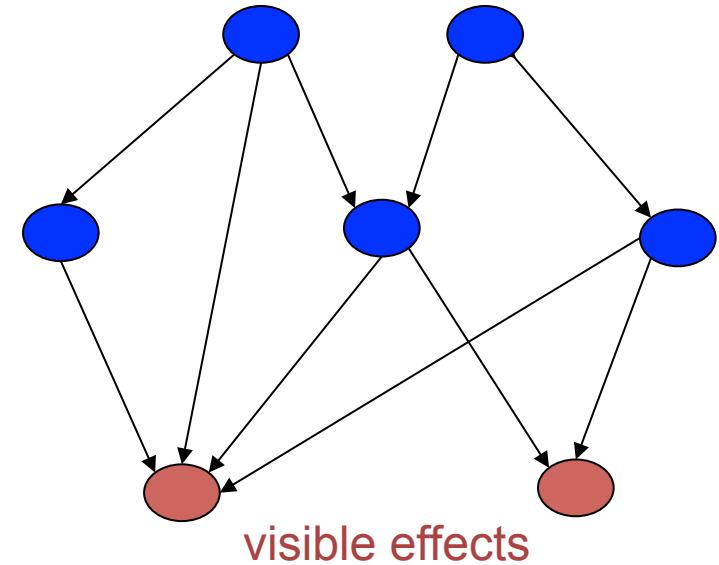
Lecture 13c Learning Sigmoid Belief Nets

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

Learning Sigmoid Belief Nets

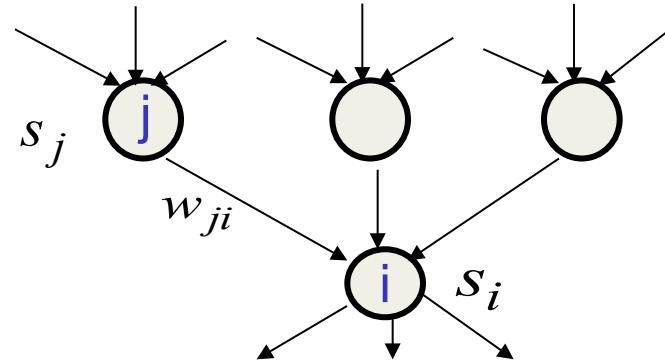
- It is easy to generate an unbiased example at the leaf nodes, so we can see what kinds of data the network believes in.
- It is hard to infer the posterior distribution over all possible configurations of hidden causes.
- It is hard to even get a sample from the posterior.
- So how can we learn sigmoid belief nets that have millions of parameters?

stochastic hidden causes



The learning rule for sigmoid belief nets

- Learning is easy if we can get an unbiased sample from the posterior distribution over hidden states given the observed data.
- For each unit, maximize the log prob. that its binary state in the sample from the posterior would be generated by the sampled binary states of its parents.

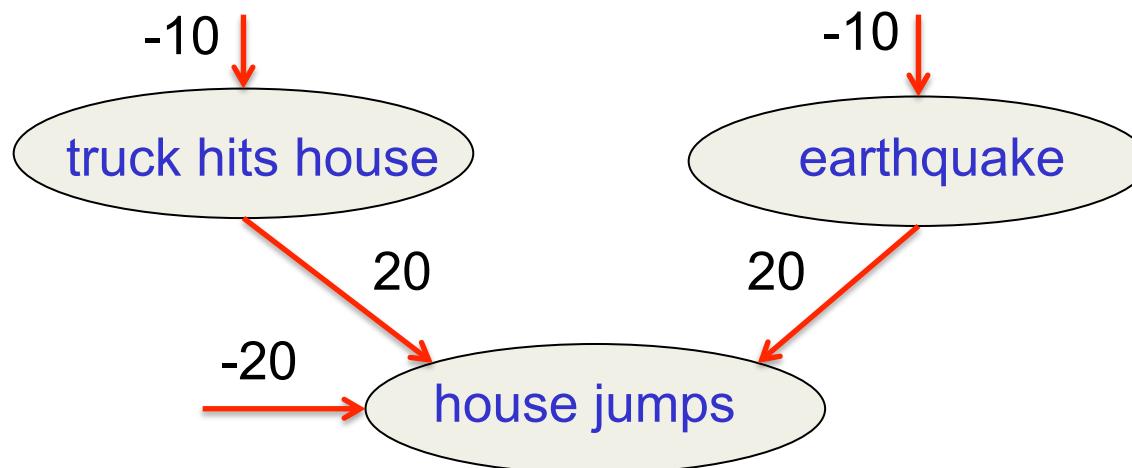


$$p_i \equiv p(s_i = 1) = \frac{1}{1 + \exp\left(-b_i - \sum_j s_j w_{ji}\right)}$$

$$\Delta w_{ji} = \epsilon s_j (s_i - p_i)$$

Explaining away (Judea Pearl)

- Even if two hidden causes are independent in the prior, they can become dependent when we observe an effect that they can both influence.
 - If we learn that there was an earthquake it reduces the probability that the house jumped because of a truck.

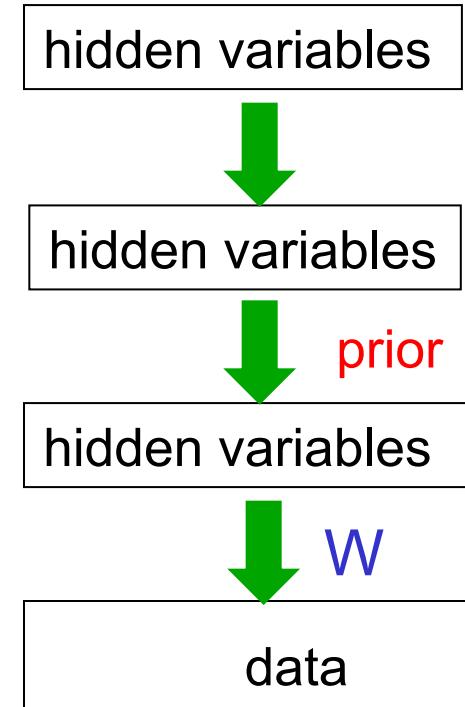


posterior
over hiddens

$$\begin{aligned} p(1,1) &= .0001 \\ p(1,0) &= .4999 \\ p(0,1) &= .4999 \\ p(0,0) &= .0001 \end{aligned}$$

Why it's hard to learn sigmoid belief nets one layer at a time

- To learn W , we need to sample from the posterior distribution in the first hidden layer.
- **Problem 1:** The posterior is not factorial because of “explaining away”.
- **Problem 2:** The posterior depends on the prior as well as the likelihood.
 - So to learn W , we need to know the weights in higher layers, even if we are only approximating the posterior. All the weights interact.
- **Problem 3:** We need to integrate over all possible configurations in the higher layers to get the prior for first hidden layer. Its hopeless!



Some methods for learning deep belief nets

- Monte Carlo methods can be used to sample from the posterior (Neal 1992).
 - But its painfully slow for large, deep belief nets.
- In the 1990's people developed variational methods for learning deep belief nets.
 - These only get approximate samples from the posterior.
- Learning with samples from the wrong distribution:
 - Maximum likelihood learning requires unbiased samples from the posterior.
- What happens if we sample from the wrong distribution but still use the maximum likelihood learning rule?
 - Does the learning still work or does it do crazy things?

Neural Networks for Machine Learning

Lecture 13d The wake-sleep algorithm

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

An apparently crazy idea

- It's hard to learn complicated models like Sigmoid Belief Nets.
- The problem is that it's hard to infer the posterior distribution over hidden configurations when given a datavector.
 - Its hard even to get a sample from the posterior.
- Crazy idea: do the inference wrong.
 - Maybe learning will still work.
 - This turns out to be true for SBNs.
- At each hidden layer, we assume (wrongly) that the posterior over hidden configurations factorizes into a product of distributions for each separate hidden unit.

Factorial distributions

- In a factorial distribution, the probability of a whole vector is just the product of the probabilities of its individual terms:

individual probabilities of
three hidden units in a layer

$$\rightarrow 0.3 \quad 0.6 \quad 0.8$$

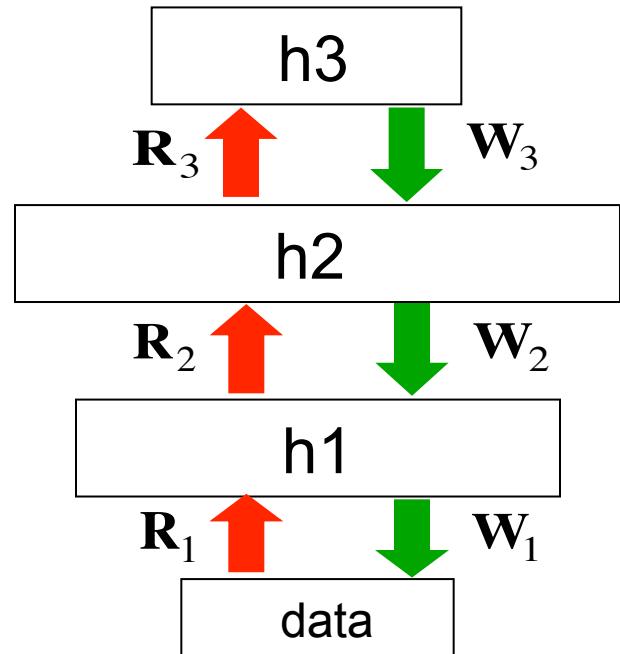
probability that the hidden
units have state 1,0,1 if the
distribution is factorial.

$$\rightarrow p(1, 0, 1) = 0.3 \times (1 - 0.6) \times 0.8$$

- A general distribution over binary vectors of length N has 2^N degrees of freedom (actually $2^N - 1$ because the probabilities must add to 1). A factorial distribution only has N degrees of freedom.

The wake-sleep algorithm (Hinton et. al. 1995)

- **Wake phase:** Use **recognition weights** to perform a bottom-up pass.
 - Train the generative weights to reconstruct activities in each layer from the layer above.
- **Sleep phase:** Use **generative weights** to generate samples from the model.
 - Train the recognition weights to reconstruct activities in each layer from the layer below.

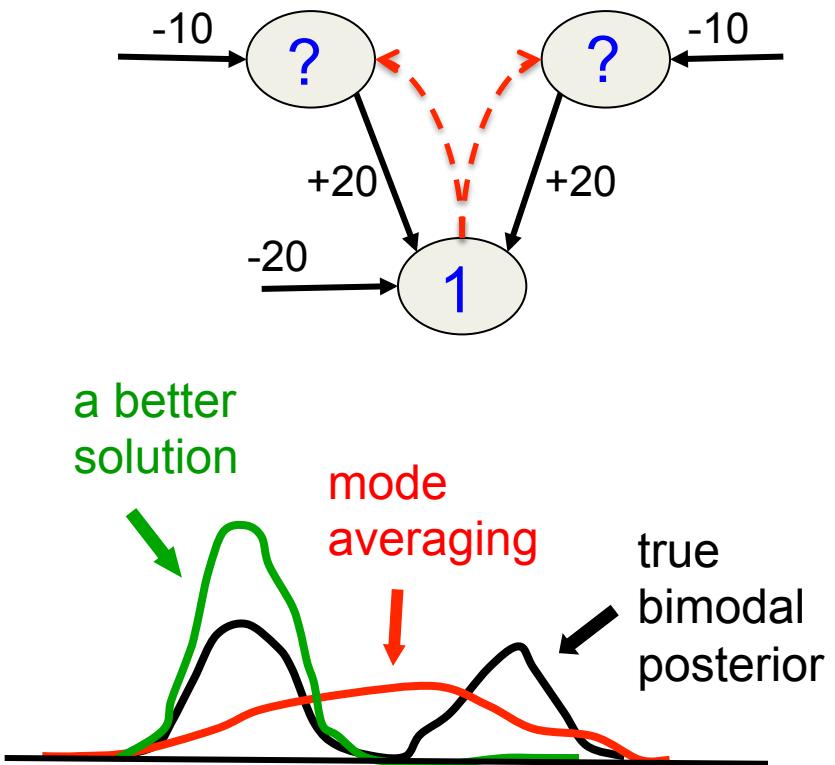


The flaws in the wake-sleep algorithm

- The recognition weights are trained to invert the generative model in parts of the space where there is no data.
 - This is wasteful.
- The recognition weights do not follow the gradient of the log probability of the data. They only approximately follow the gradient of the variational bound on this probability.
 - This leads to incorrect mode-averaging
- The posterior over the top hidden layer is very far from independent because of explaining away effects.
- Nevertheless, Karl Friston thinks this is how the brain works.

Mode averaging

- If we generate from the model, half the instances of a 1 at the data layer will be caused by a (1,0) at the hidden layer and half will be caused by a (0,1).
 - So the **recognition weights** will learn to produce (0.5, 0.5)
 - This represents a distribution that puts half its mass on 1,1 or 0,0: very improbable hidden configurations.
- Its much better to just pick one mode.
 - This is the best recognition model you can get if you assume that the posterior over hidden states factorizes.



Neural Networks for Machine Learning

Lecture 14a

Learning layers of features by stacking RBMs

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

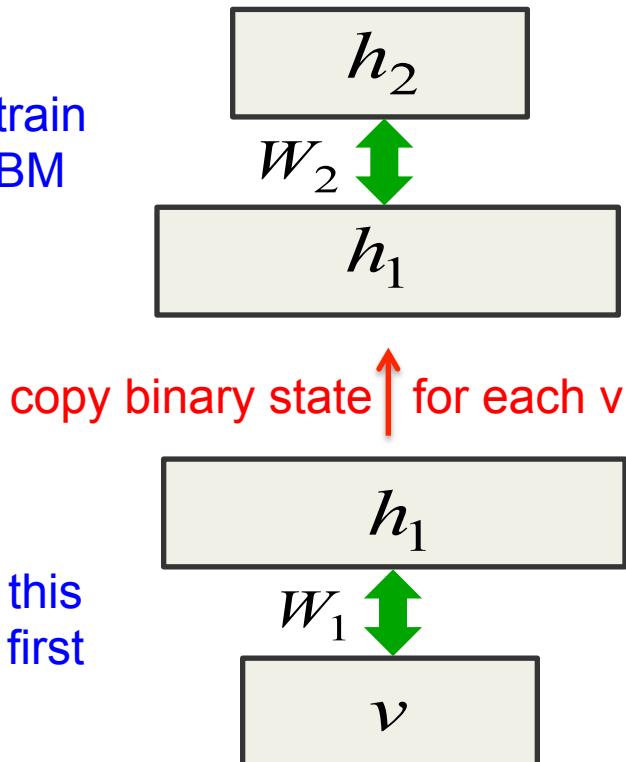
Abdel-rahman Mohamed

Training a deep network by stacking RBMs

- First train a layer of features that receive input directly from the pixels.
- Then treat the activations of the trained features as if they were pixels and learn features of features in a second hidden layer.
- Then do it again.
- It can be proved that each time we add another layer of features we improve a variational lower bound on the log probability of generating the training data.
 - The proof is complicated and only applies to unreal cases.
 - It is based on a neat equivalence between an RBM and an infinitely deep belief net (see lecture 14b).

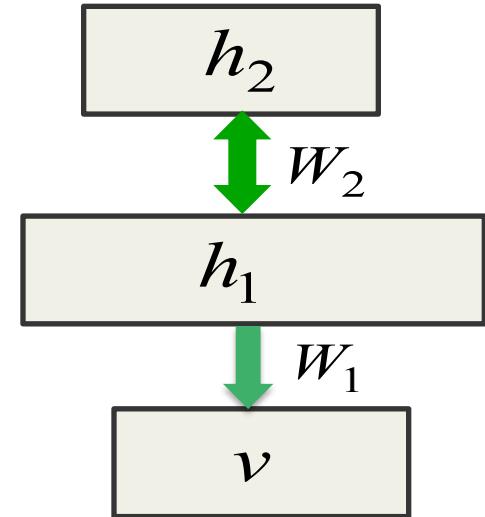
Combining two RBMs to make a DBN

Then train
this RBM



Train this
RBM first

Compose the
two RBM
models to
make a single
DBN model



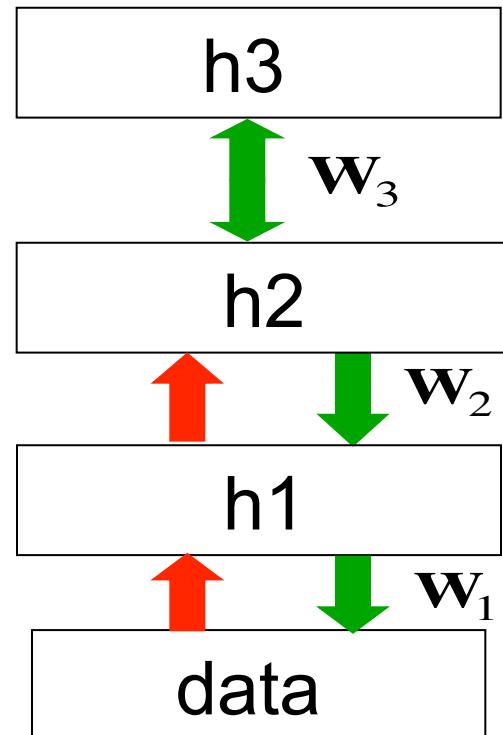
It's not a Boltzmann machine!

The generative model after learning 3 layers

To generate data:

1. Get an equilibrium sample from the top-level RBM by performing alternating Gibbs sampling for a long time.
2. Perform a top-down pass to get states for all the other layers.

The lower level bottom-up connections are **not** part of the generative model.
They are just used for inference.



An aside: Averaging factorial distributions

- If you average some factorial distributions, you do NOT get a factorial distribution.
 - In an RBM, the posterior over 4 hidden units is factorial **for each visible vector.**
- Posterior for v1: 0.9, 0.9, 0.1, 0.1
- Posterior for v2: $\frac{0.1, 0.1, 0.9, 0.9}{0.5, 0.5, 0.5, 0.5}$
- Consider the binary vector 1,1,0,0.
 - in the posterior for v1, $p(1,1,0,0) = 0.9^4 = 0.43$
 - in the posterior for v2, $p(1,1,0,0) = 0.1^4 = .0001$
 - in the aggregated posterior, $p(1,1,0,0) = 0.215.$
- If the aggregated posterior was factorial it would have $p = 0.5^4$

Why does greedy learning work?

The weights, W , in the bottom level RBM define many different distributions: $p(v|h)$; $p(h|v)$; $p(v,h)$; $p(h)$; $p(v)$.

We can express the RBM model as
$$p(v) = \sum_h p(h) p(v | h)$$

If we leave $p(v|h)$ alone and improve $p(h)$, we will improve $p(v)$.

To improve $p(h)$, we need it to be a better model than $p(h;W)$ of the **aggregated posterior** distribution over hidden vectors produced by applying W transpose to the data.

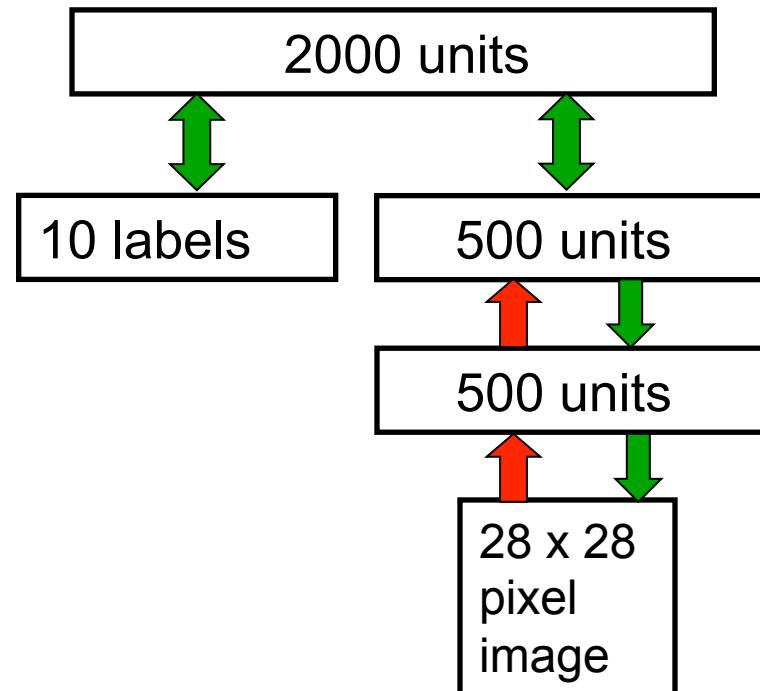
Fine-tuning with a contrastive version of the wake-sleep algorithm

After learning many layers of features, we can fine-tune the features to improve generation.

1. Do a stochastic bottom-up pass
 - Then adjust the top-down weights of lower layers to be good at reconstructing the feature activities in the layer below.
2. Do a few iterations of sampling in the top level RBM
 - Then adjust the weights in the top-level RBM using CD.
3. Do a stochastic top-down pass
 - Then Adjust the bottom-up weights to be good at reconstructing the feature activities in the layer above.

The DBN used for modeling the joint distribution of MNIST digits and their labels

- The first two hidden layers are learned without using labels.
- The top layer is learned as an RBM for modeling the labels concatenated with the features in the second hidden layer.
- The weights are then fine-tuned to be a better generative model using contrastive wake-sleep.



Neural Networks for Machine Learning

Lecture 14b Discriminative fine-tuning for DBNs

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

Fine-tuning for discrimination

- First learn one layer at a time by stacking RBMs.
- Treat this as “pre-training” that finds a good initial set of weights which can then be fine-tuned by a local search procedure.
 - Contrastive wake-sleep is a way of fine-tuning the model to be better at generation.
- Backpropagation can be used to fine-tune the model to be better at **discrimination**.
 - This overcomes many of the limitations of standard backpropagation.
 - It makes it easier to learn deep nets.
 - It makes the nets generalize better.

Why backpropagation works better with greedy pre-training: The optimization view

- Greedily learning one layer at a time scales well to really big networks, especially if we have locality in each layer.
- We do not start backpropagation until we already have sensible feature detectors that should already be very helpful for the discrimination task.
 - So the initial gradients are sensible and backpropagation only needs to perform a local search from a sensible starting point.

Why backpropagation works better with greedy pre-training: The overfitting view

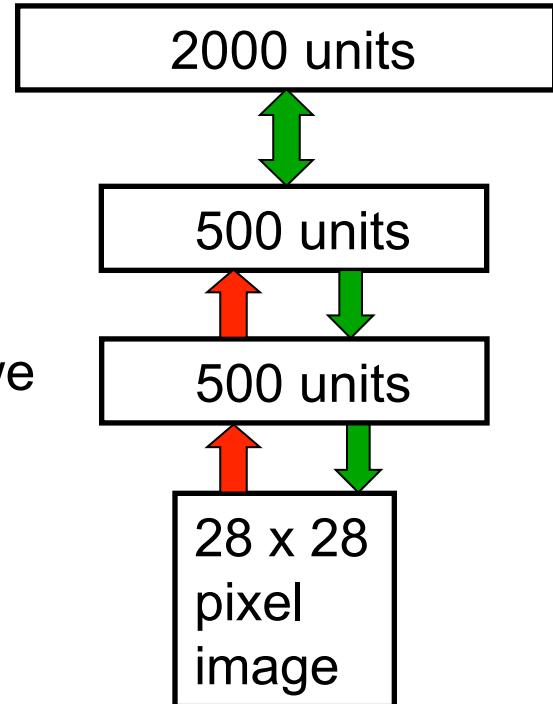
- Most of the information in the final weights comes from modeling the distribution of input vectors.
 - The input vectors generally contain a lot more information than the labels.
 - The precious information in the labels is only used for the fine-tuning.
- The fine-tuning only modifies the features slightly to get the category boundaries right. It does not need to discover new features.
- This type of back-propagation works well even if most of the training data is unlabeled.
 - The unlabeled data is still very useful for discovering good features.
- An objection: Surely, many of the features will be useless for any particular discriminative task (consider shape & pose).
 - But the ones that are useful will be much more useful than the raw inputs.

First, model the distribution of digit images

The top two layers form a restricted Boltzmann machine whose energy landscape should model the low dimensional manifolds of the digits.

The network learns a density model for unlabeled digit images. When we generate from the model we get things that look like real digits of all classes.

But do the hidden features really help with digit discrimination? Add a 10-way softmax at the top and do backpropagation.



Results on the permutation-invariant MNIST task

	Error rate
• Backprop net with one or two hidden layers (Platt; Hinton)	1.6%
• Backprop with L2 constraints on incoming weights	1.5%
• Support Vector Machines (Decoste & Schoelkopf, 2002)	1.4%
• Generative model of joint density of images and labels (+ generative fine-tuning)	1.25%
• Generative model of unlabelled digits followed by gentle backpropagation (Hinton & Salakhutdinov, 2006)	1.15% → 1.0%

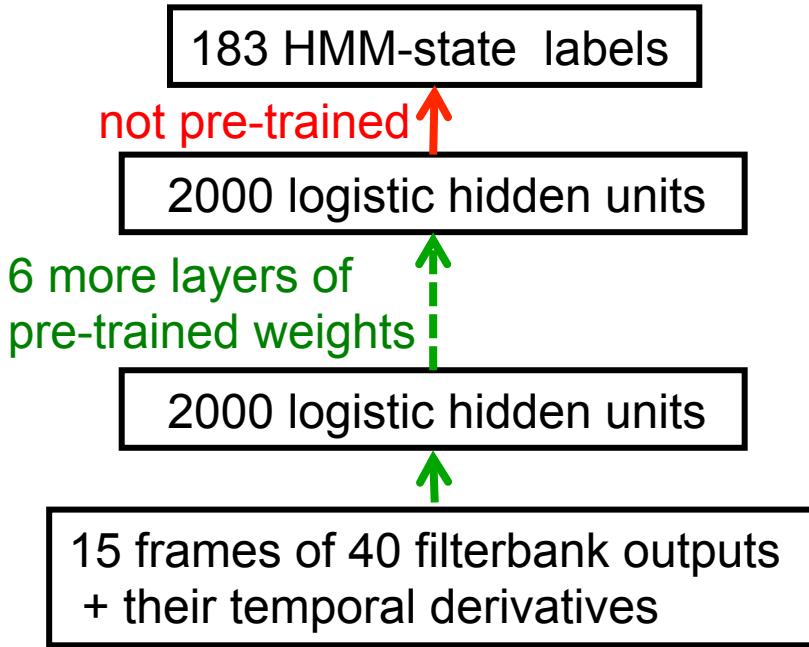
Unsupervised “pre-training” also helps for models that have more data and better priors

- Ranzato et. al. (NIPS 2006) used an additional 600,000 distorted digits.
- They also used convolutional multilayer neural networks.

Back-propagation alone: 0.49%

Unsupervised layer-by-layer
pre-training followed by backprop: 0.39% (record at the time)

Phone recognition on the TIMIT benchmark (Mohamed, Dahl, & Hinton, 2009 & 2012)



- After standard post-processing using a bi-phone model, a deep net with 8 layers gets **20.7%** error rate.
- The best previous speaker-independent result on TIMIT was **24.4%** and this required averaging several models.
- Li Deng (at MSR) realised that this result could change the way speech recognition was done. It has!

<http://www.bbc.co.uk/news/technology-20266427>

Neural Networks for Machine Learning

Lecture 14c

What happens during discriminative fine-tuning?

Geoffrey Hinton

Nitish Srivastava,

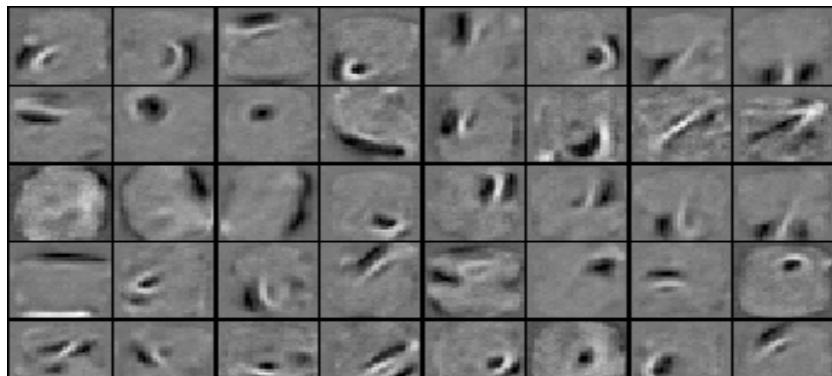
Kevin Swersky

Tijmen Tieleman

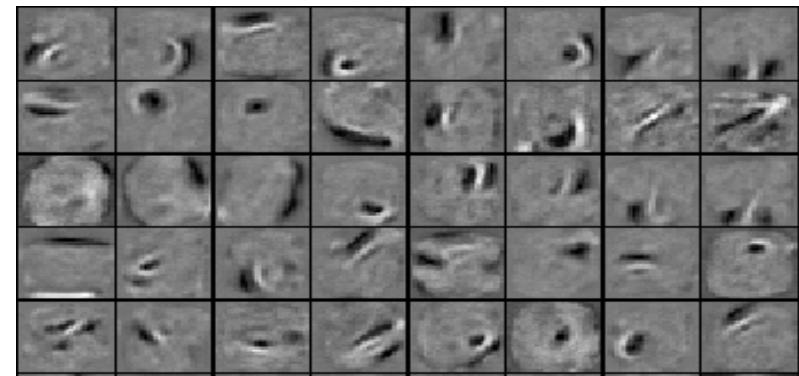
Abdel-rahman Mohamed

Learning Dynamics of Deep Nets

the next 4 slides describe work by Yoshua Bengio's group



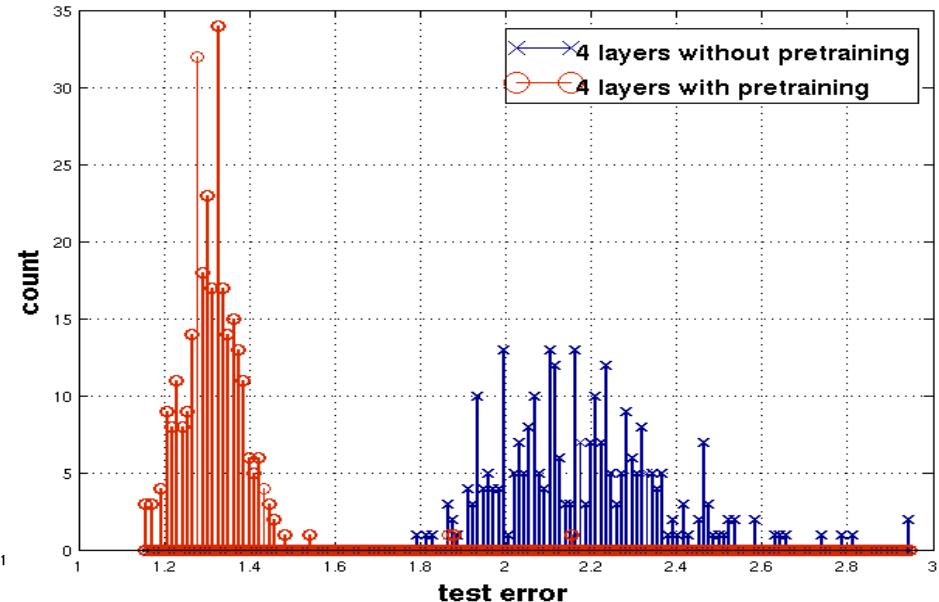
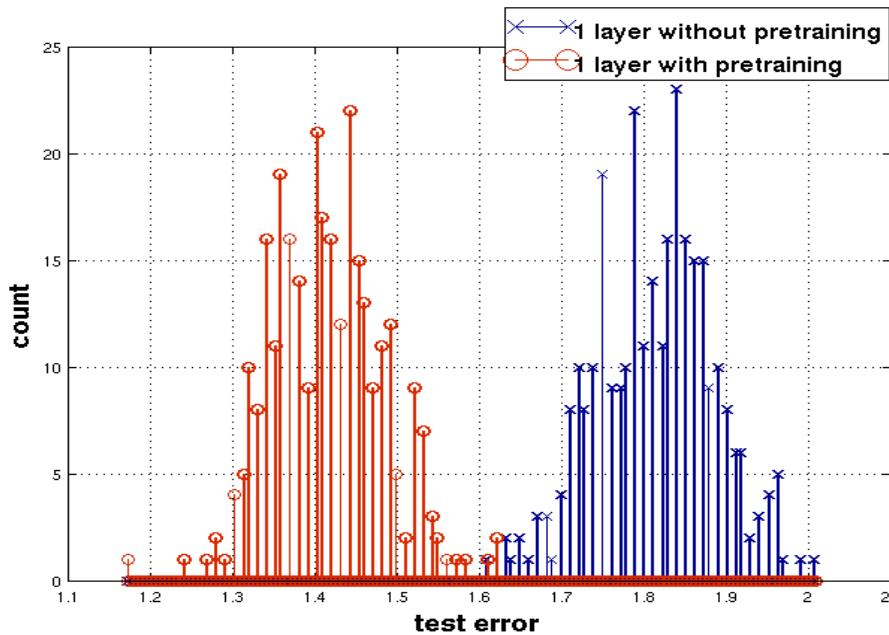
Before fine-tuning



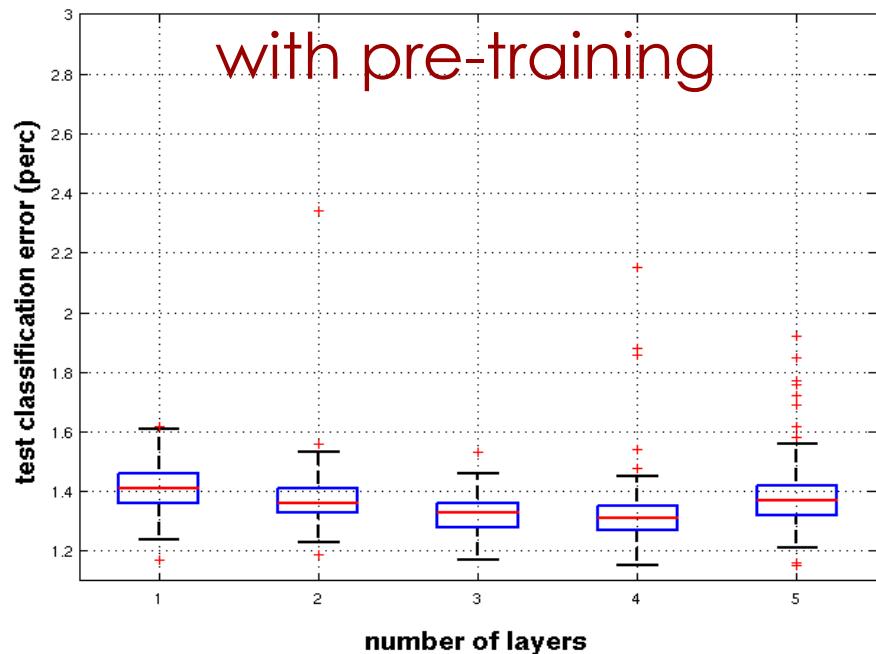
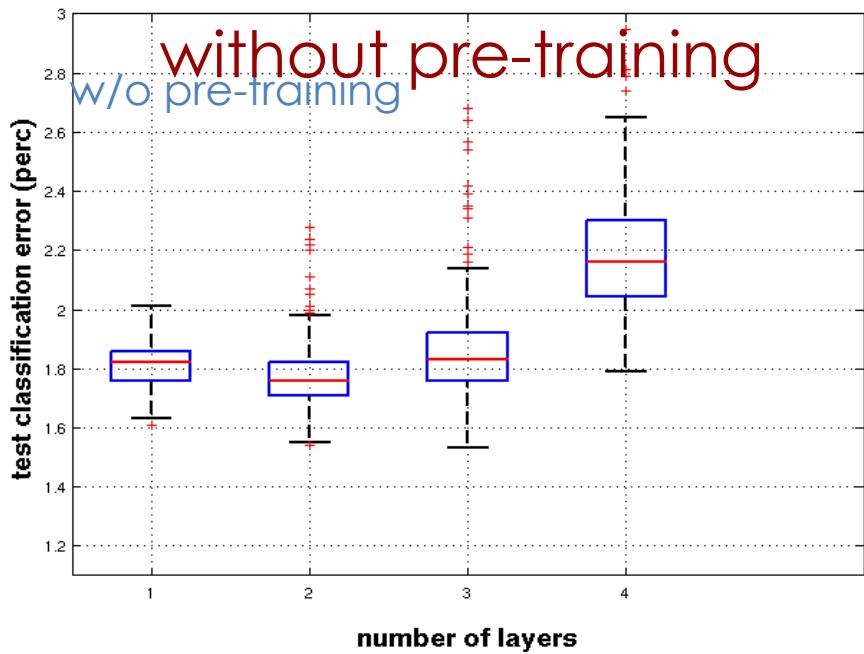
After fine-tuning

Effect of Unsupervised Pre-training

Erhan et. al. AISTATS' 2009



Effect of Depth

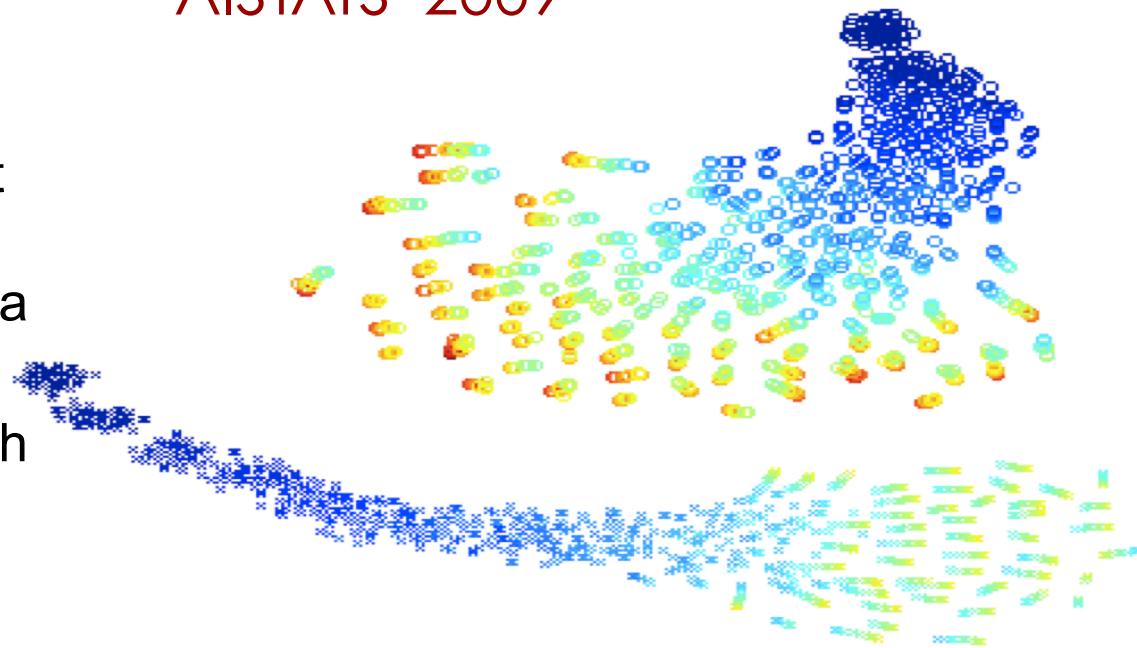


Trajectories of the learning in function space

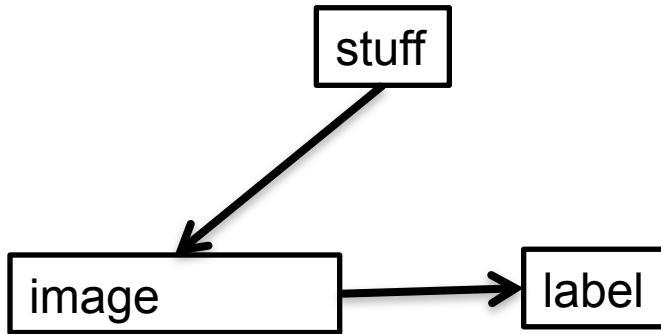
(a 2-D visualization produced with t-SNE)

- Each point is a model in function space
- Color = epoch
- Top: trajectories without pre-training. Each trajectory converges to a different local min.
- Bottom: Trajectories with pre-training.
- No overlap!

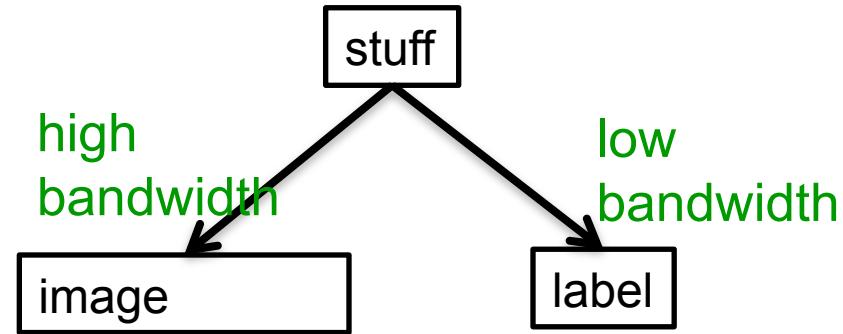
Erhan et. al
AISTATS' 2009



Why unsupervised pre-training makes sense



If image-label pairs were generated this way, it would make sense to try to go straight from images to labels. For example, do the pixels have even parity?



If image-label pairs are generated this way, it makes sense to first learn to recover the stuff that caused the image by inverting the high bandwidth pathway.

Neural Networks for Machine Learning

Lecture 14d Modeling real-valued data with an RBM

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

Modeling real-valued data

- For images of digits, intermediate intensities can be represented as if they were probabilities by using “mean-field” logistic units.
 - We treat intermediate values as the probability that the pixel is inked.
- This will not work for real images.
 - In a real image, the intensity of a pixel is almost always, almost exactly the average of the neighboring pixels.
 - Mean-field logistic units cannot represent precise intermediate values.

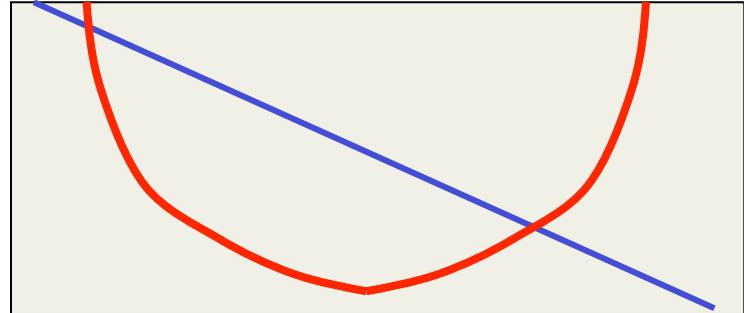
A standard type of real-valued visible unit

- Model pixels as Gaussian variables. Alternating Gibbs sampling is still easy, though learning needs to be much slower.

$$E(\mathbf{v}, \mathbf{h}) = \sum_{i \in vis} \frac{(v_i - b_i)^2}{2\sigma_i^2} - \sum_{j \in hid} b_j h_j - \sum_{i,j} \frac{v_i}{\sigma_i} h_j w_{ij}$$

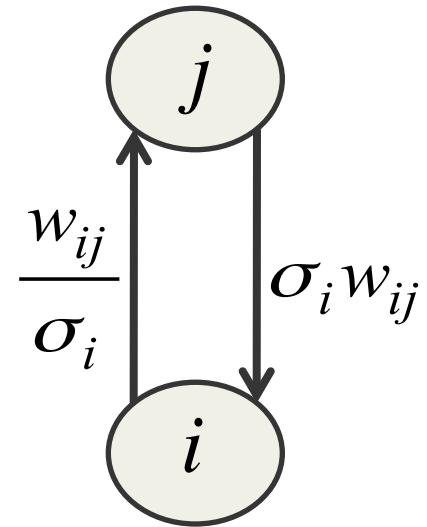
parabolic containment function

↑
energy-gradient produced by the total input to a visible unit



Gaussian-Binary RBM's

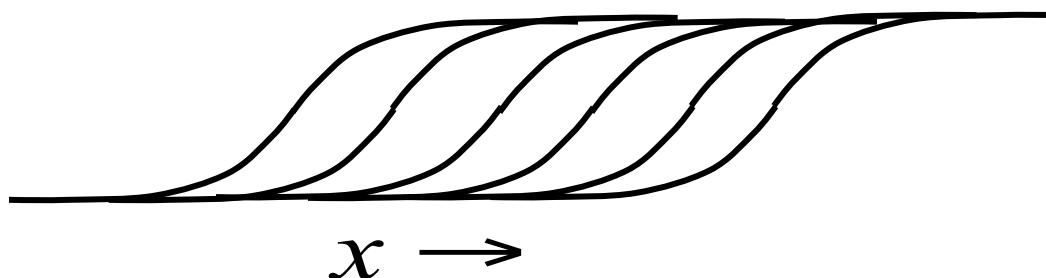
- Lots of people have failed to get these to work properly. It's extremely hard to learn tight variances for the visible units.
 - It took a long time for us to figure out why it is so hard to learn the visible variances.
- When sigma is small, we need many more hidden units than visible units.
 - This allows small weights to produce big top-down effects.



When sigma is much less than 1, the bottom-up effects are too big and the top-down effects are too small.

Stepped sigmoid units: A neat way to implement integer values

- Make many copies of a stochastic binary unit.
- All copies have the same weights and the same adaptive bias, b , but they have different fixed offsets to the bias:
 $b - 0.5, b - 1.5, b - 2.5, b - 3.5, \dots$



Fast approximations



$$\langle y \rangle = \sum_{n=1}^{n=\infty} \sigma(x + 0.5 - n) \approx \log(1 + e^x) \approx \max(0, x + \text{noise})$$

- Contrastive divergence learning works well for the sum of stochastic logistic units with offset biases. The noise variance is $\sigma(y)$
- It also works for rectified linear units. These are much faster to compute than the sum of many logistic units with different biases.

A nice property of rectified linear units

- If a relu has a bias of zero, it exhibits scale equivariance:
 - This is a very nice property to have for images.

$$R(a \mathbf{x}) = a R(\mathbf{x}) \quad \text{but} \quad R(a + b) \neq R(a) + R(b)$$

- It is like the equivariance to translation exhibited by convolutional nets.

$$R(shift(\mathbf{x})) = shift(R(\mathbf{x}))$$

Neural Networks for Machine Learning

Lecture 14e

RBMs are Infinite Sigmoid Belief Nets

ADVANCED MATERIAL: NOT ON QUIZZES OR FINAL TEST

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

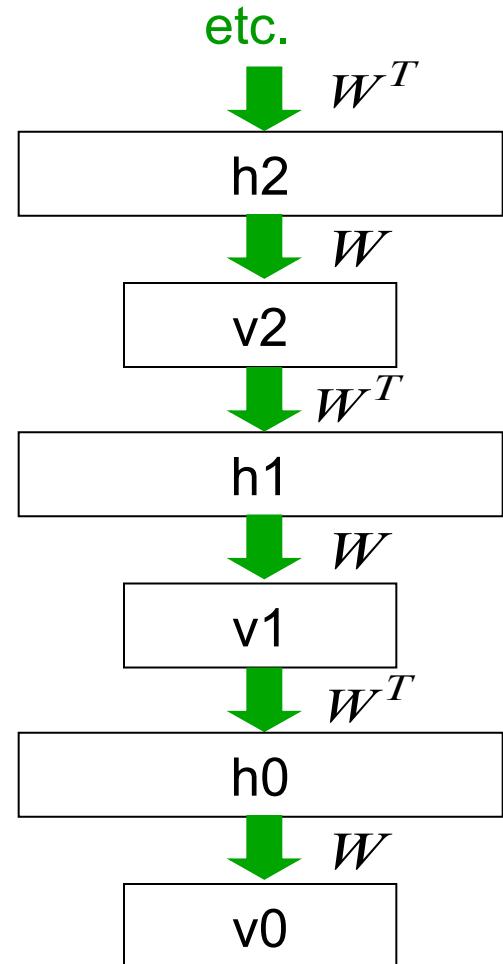
Abdel-rahman Mohamed

Another view of why layer-by-layer learning works (Hinton, Osindero & Teh 2006)

- There is an unexpected equivalence between RBM's and directed networks with many layers that all share the same weight matrix.
 - This equivalence also gives insight into why contrastive divergence learning works.
- An RBM is actually just an infinitely deep sigmoid belief net with a lot of weight sharing.
 - The Markov chain we run when we want to sample from the equilibrium distribution of an RBM can be viewed as a sigmoid belief net.

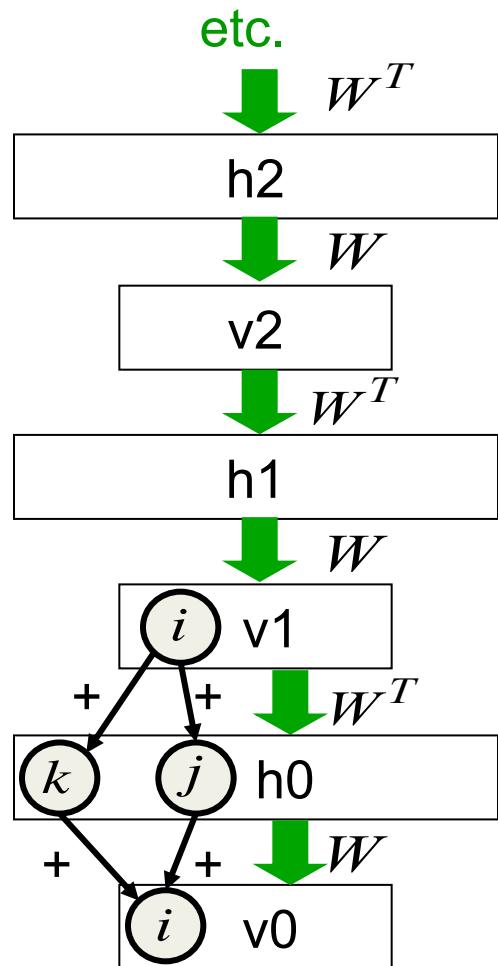
An infinite sigmoid belief net that is equivalent to an RBM

- The distribution generated by this infinite directed net with replicated weights is the equilibrium distribution for a compatible pair of conditional distributions: $p(v|h)$ and $p(h|v)$ that are both defined by W
 - A top-down pass of the directed net is exactly equivalent to letting a Restricted Boltzmann Machine settle to equilibrium.
 - So this infinite directed net defines the same distribution as an RBM.



Inference in an infinite sigmoid belief net

- The variables in h_0 are conditionally independent given v_0 .
 - Inference is trivial. Just multiply v_0 by W^T
 - The model above h_0 implements a complementary prior.
 - Multiplying v_0 by W^T gives the product of the likelihood term and the prior term.
 - The complementary prior cancels the explaining away.
- Inference in the directed net is exactly equivalent to letting an RBM settle to equilibrium starting at the data.



- The learning rule for a sigmoid belief net is:

$$\Delta w_{ij} \propto s_j(s_i - p_i)$$

s_i^1 is an unbiased sample from p_i^0

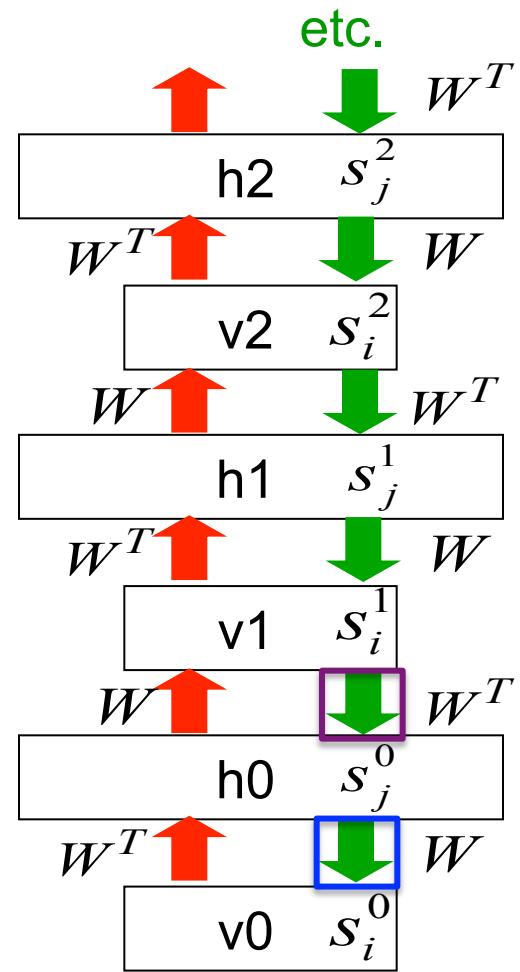
- With replicated weights this rule becomes:

$$s_j^0(s_i^0 - s_i^1) +$$

$$s_i^1(s_j^0 - s_j^1) +$$

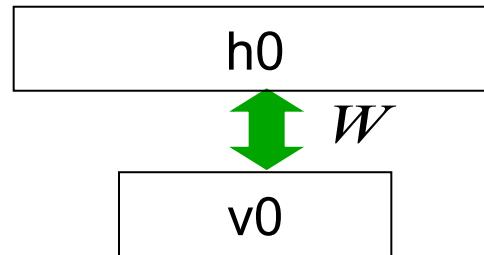
$$s_j^1(s_i^1 - s_i^2) + \dots$$

$$- s_j^\infty s_i^\infty$$

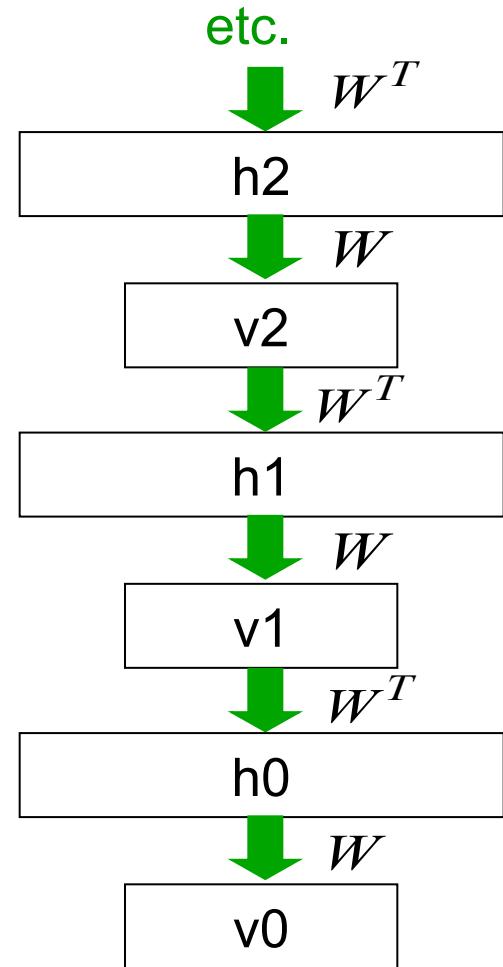


Learning a deep directed network

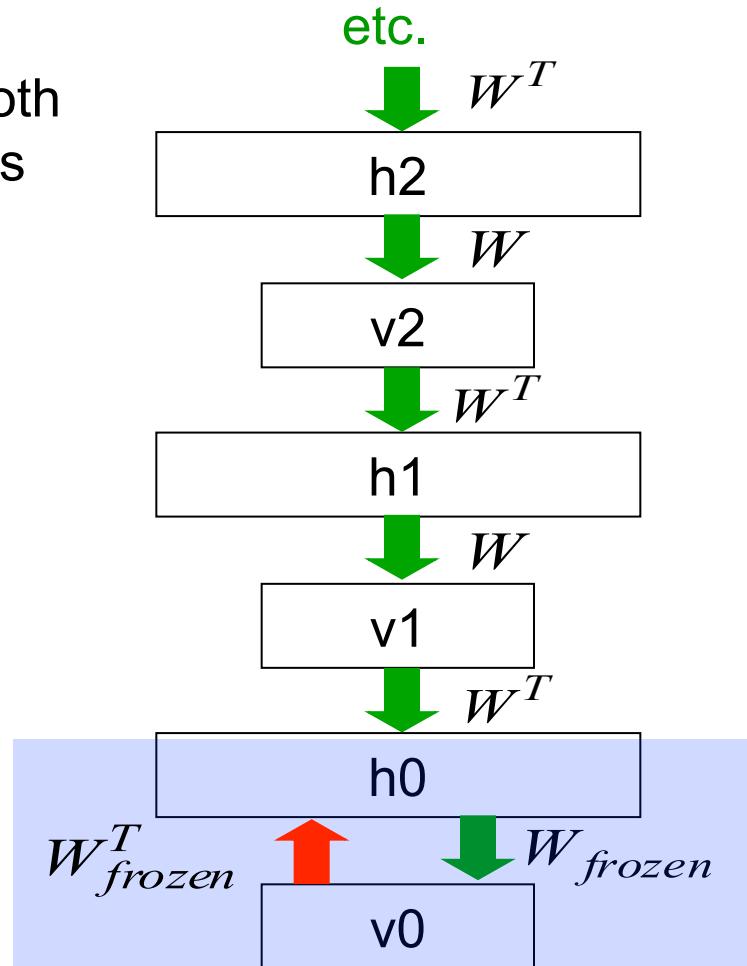
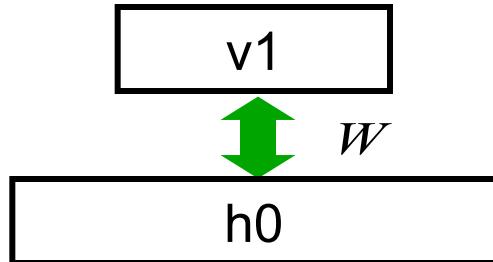
- First learn with all the weights tied. This is exactly equivalent to learning an RBM.



- Think of the symmetric connections as a shorthand notation for an infinite directed net with tied weights.
- We ought to use maximum likelihood learning, but we use CD1 as a shortcut.



- Then freeze the first layer of weights in both directions and learn the remaining weights (still tied together).
 - This is equivalent to learning another RBM, using the aggregated posterior distribution of h_0 as the data.



What happens when the weights in higher layers become different from the weights in the first layer?

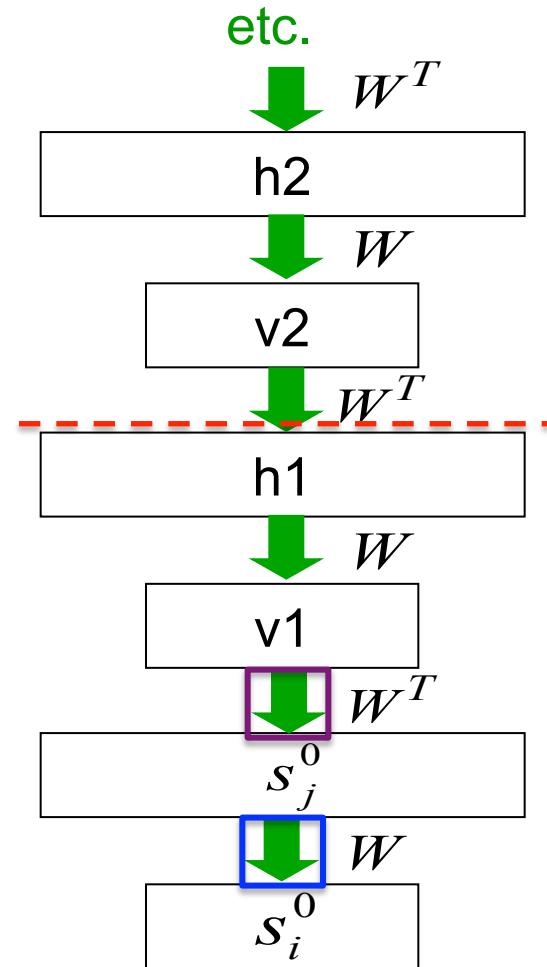
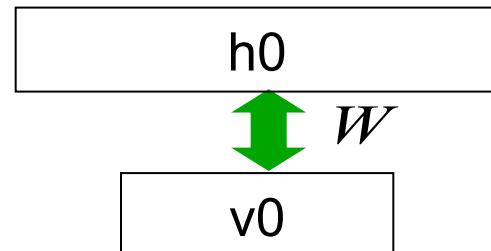
- The higher layers no longer implement a complementary prior.
 - So performing inference using the frozen weights in the first layer is no longer correct.
 - But it's still pretty good.
 - Using this incorrect inference procedure gives a variational lower bound on the log probability of the data.
- The higher layers learn a prior that is closer to the aggregated posterior distribution of the first hidden layer.
 - This improves the network's model of the data.
 - Hinton, Osindero and Teh (2006) prove that this improvement is always bigger than the loss in the variational bound caused by using less accurate inference.

What is really happening in contrastive divergence learning?

- Contrastive divergence learning in this RBM is equivalent to **ignoring** the small derivatives contributed by the tied weights in higher layers.

$$s_j^0(s_i^0 - s_i^1) +$$

$$s_i^1(s_j^0 - s_j^1) = s_j^0 s_i^0 - s_i^1 s_j^1$$



Why is it OK to ignore the derivatives in higher layers?

- When the weights are small, the Markov chain mixes fast.
 - So the higher layers will be close to the equilibrium distribution (i.e they will have “forgotten” the datavector).
 - At equilibrium the derivatives must average to zero, because the current weights are a perfect model of the equilibrium distribution!
- As the weights grow we may need to run more iterations of CD.
 - This allows CD to continue to be a good approximation to maximum likelihood.
 - But for learning layers of features, it does not need to be a good approximation to maximum likelihood!

Neural Networks for Machine Learning

Lecture 15a

From Principal Components Analysis to Autoencoders

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

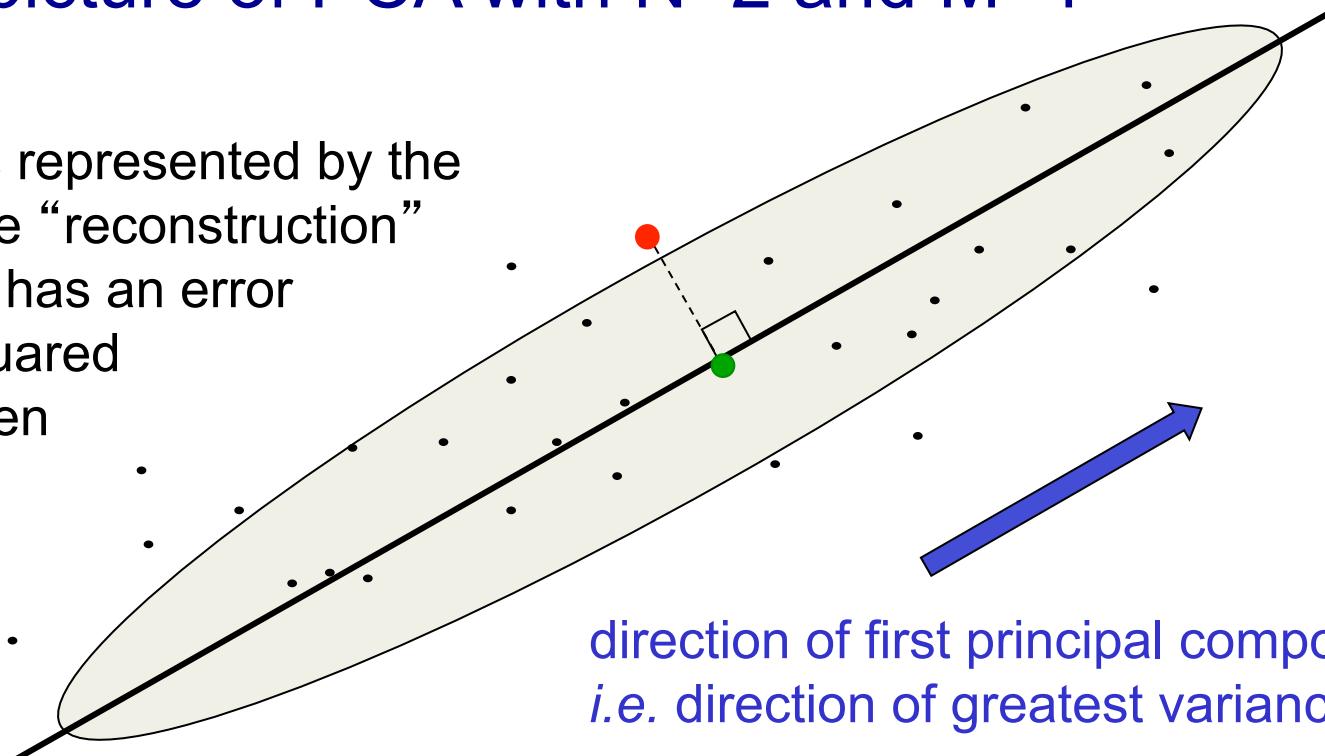
Abdel-rahman Mohamed

Principal Components Analysis

- This takes N-dimensional data and finds the M orthogonal directions in which the data have the most variance.
 - These M principal directions form a lower-dimensional subspace.
 - We can represent an N-dimensional datapoint by its projections onto the M principal directions.
 - This loses all information about where the datapoint is located in the remaining orthogonal directions.
- We reconstruct by using the mean value (over all the data) on the N-M directions that are not represented.
 - The reconstruction error is the sum over all these unrepresented directions of the squared differences of the datapoint from the mean.

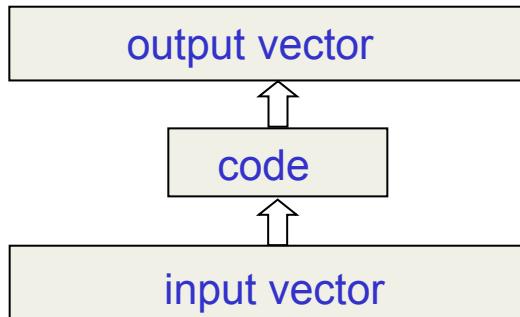
A picture of PCA with N=2 and M=1

The red point is represented by the green point. The “reconstruction” of the red point has an error equal to the squared distance between red and green points.



Using backpropagation to implement PCA inefficiently

- Try to make the output be the same as the input in a network with a central bottleneck.

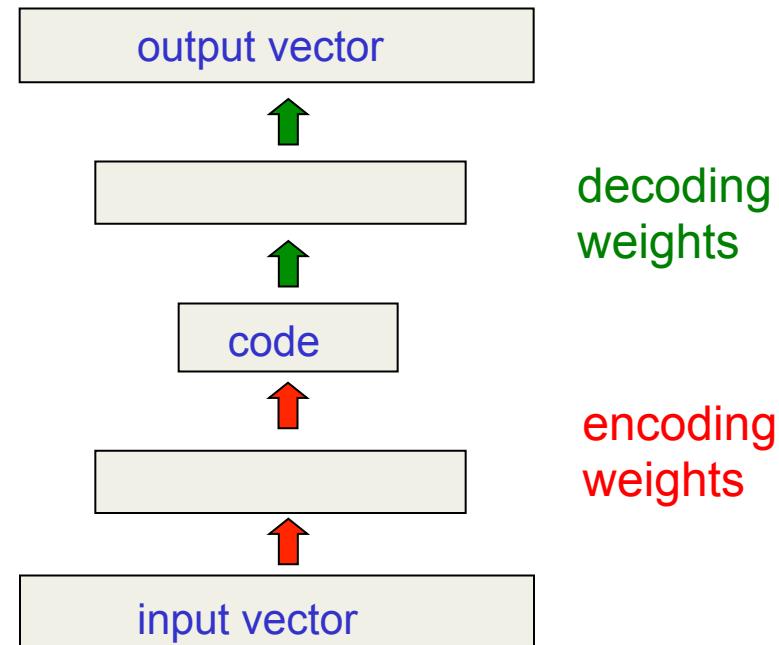


- The activities of the hidden units in the bottleneck form an efficient code.

- If the hidden and output layers are linear, it will learn hidden units that are a linear function of the data and minimize the squared reconstruction error.
 - This is exactly what PCA does.
- The M hidden units will span the same space as the first M components found by PCA
 - Their weight vectors may not be orthogonal.
 - They will tend to have equal variances.

Using backpropagation to generalize PCA

- With non-linear layers before and after the code, it should be possible to efficiently represent data that lies on or near a non-linear manifold.
 - The encoder converts coordinates in the input space to coordinates on the manifold.
 - The decoder does the inverse mapping.



Neural Networks for Machine Learning

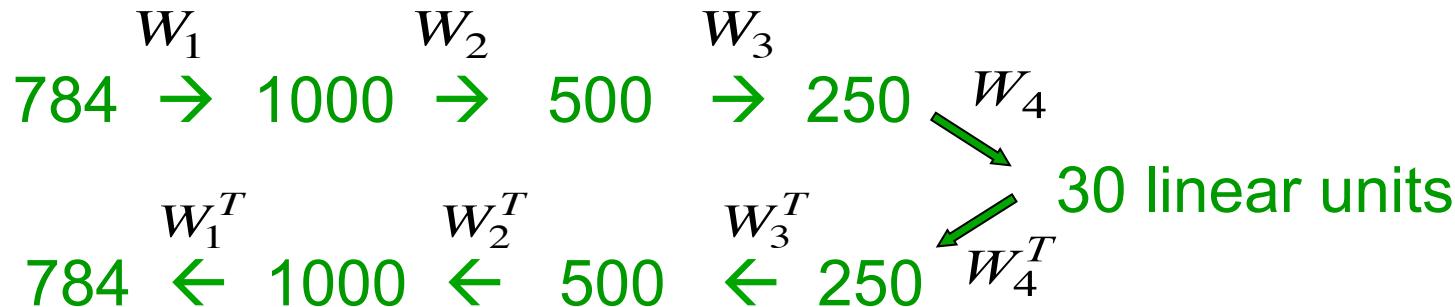
Lecture 15b Deep Autoencoders

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

Deep Autoencoders

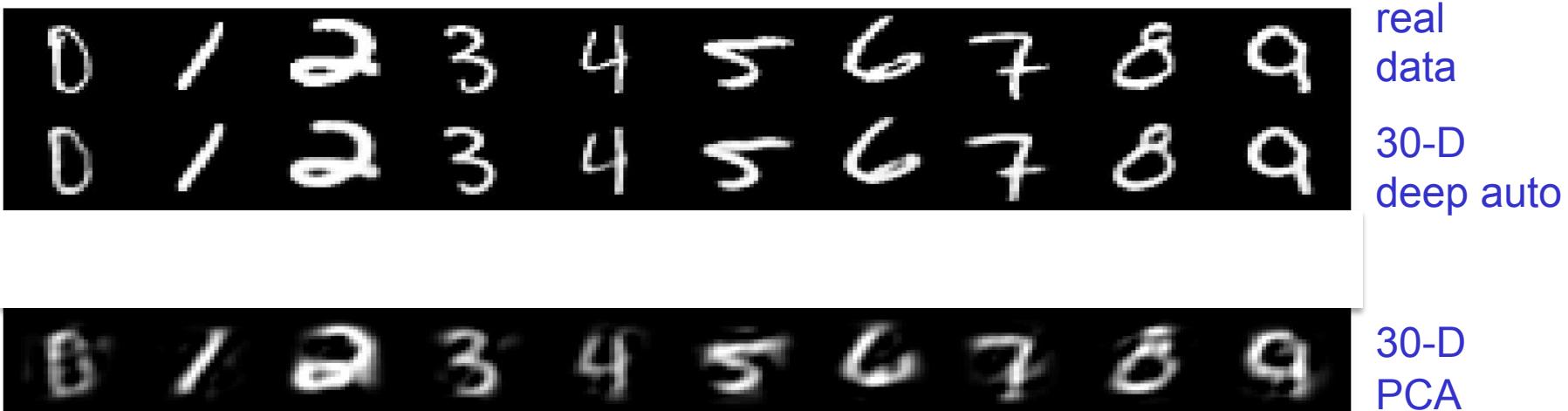
- They always looked like a really nice way to do non-linear dimensionality reduction:
 - They provide flexible mappings both ways.
 - The learning time is linear (or better) in the number of training cases.
 - The final encoding model is fairly compact and fast.
- But it turned out to be very difficult to optimize deep autoencoders using backpropagation.
 - With small initial weights the backpropagated gradient dies.
- We now have a much better ways to optimize them.
 - Use unsupervised layer-by-layer pre-training.
 - Or just initialize the weights carefully as in Echo-State Nets.

The first really successful deep autoencoders (Hinton & Salakhutdinov, Science, 2006)



We train a stack of 4 RBM's and then “unroll” them.
Then we fine-tune with gentle backprop.

A comparison of methods for compressing digit images to 30 real numbers



Neural Networks for Machine Learning

Lecture 15c

Deep autoencoders for document retrieval and visualization

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

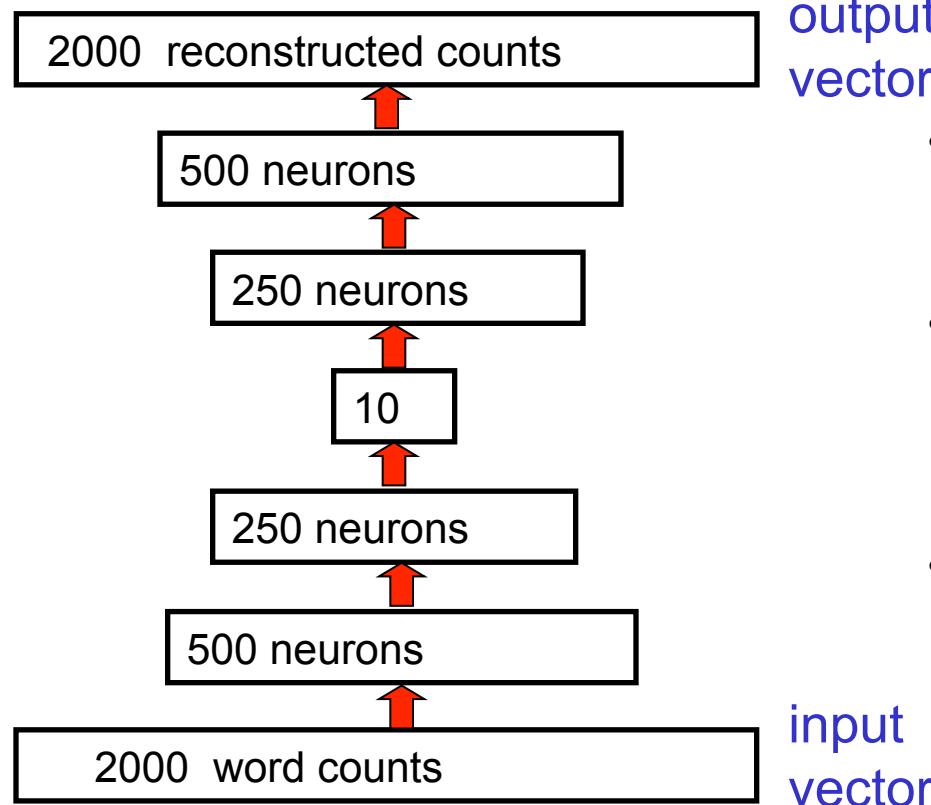
Abdel-rahman Mohamed

How to find documents that are similar to a query document

- Convert each document into a “bag of words”.
 - This is a vector of word counts ignoring order.
 - Ignore stop words (like “the” or “over”)
- We could compare the word counts of the query document and millions of other documents but this is too slow.
 - So we reduce each query vector to a much smaller vector that still contains most of the information about the content of the document.

0	fish
0	cheese
2	vector
2	count
0	school
2	query
1	reduce
1	bag
0	pulpit
0	iraq
2	word

How to compress the count vector



- We train the neural network to reproduce its input vector as its output
- This forces it to compress as much information as possible into the 10 numbers in the central bottleneck.
- These 10 numbers are then a good way to compare documents.

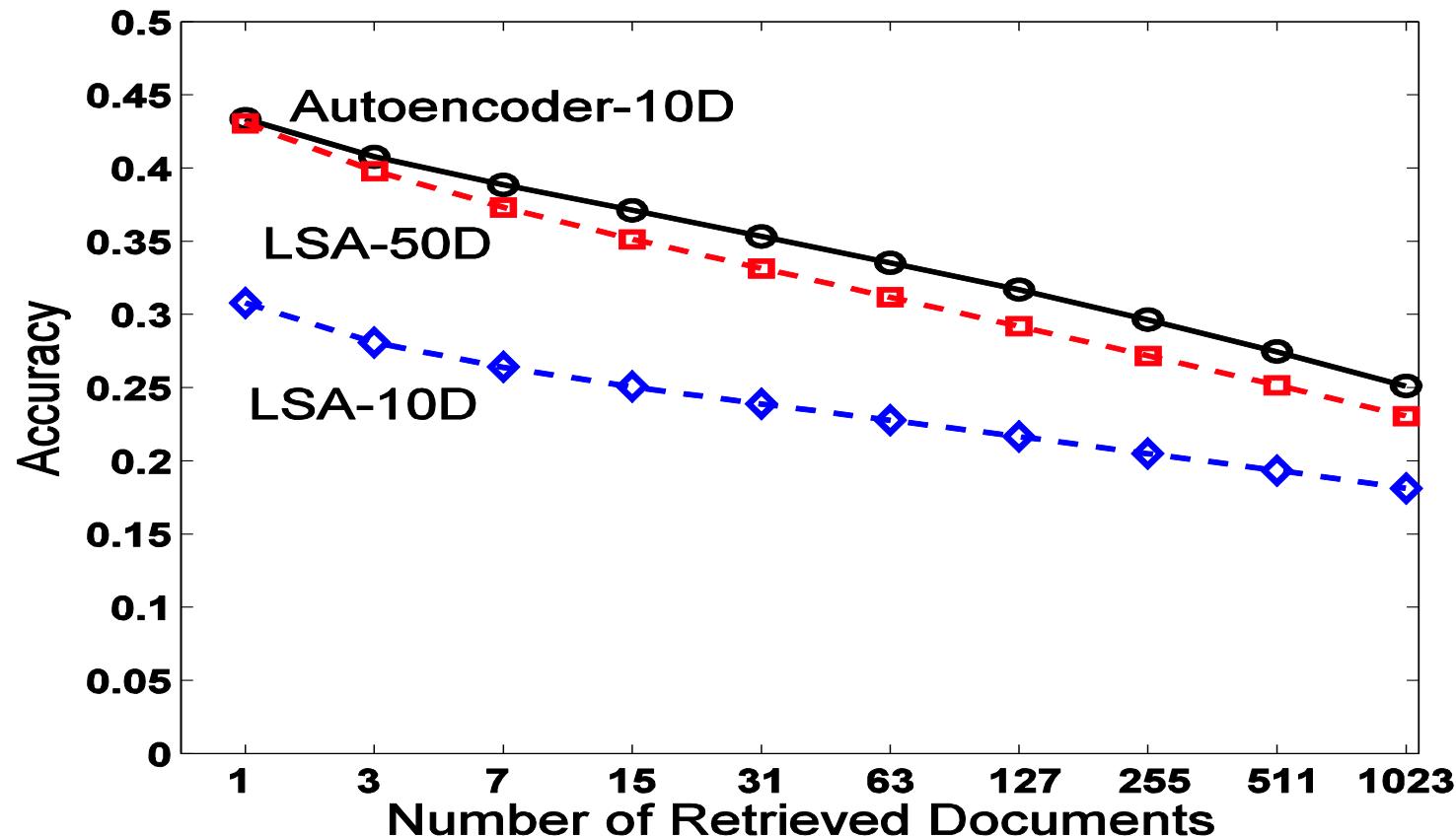
The non-linearity used for reconstructing bags of words

- Divide the counts in a bag of words vector by N , where N is the total number of non-stop words in the document.
 - The resulting probability vector gives the probability of getting a particular word if we pick a non-stop word at random from the document.
- At the output of the autoencoder, we use a softmax.
 - The probability vector defines the desired outputs of the softmax.
- When we train the first RBM in the stack we use the same trick.
 - We treat the word counts as probabilities, but we make the visible to hidden weights N times bigger than the hidden to visible because we have N observations from the probability distribution.

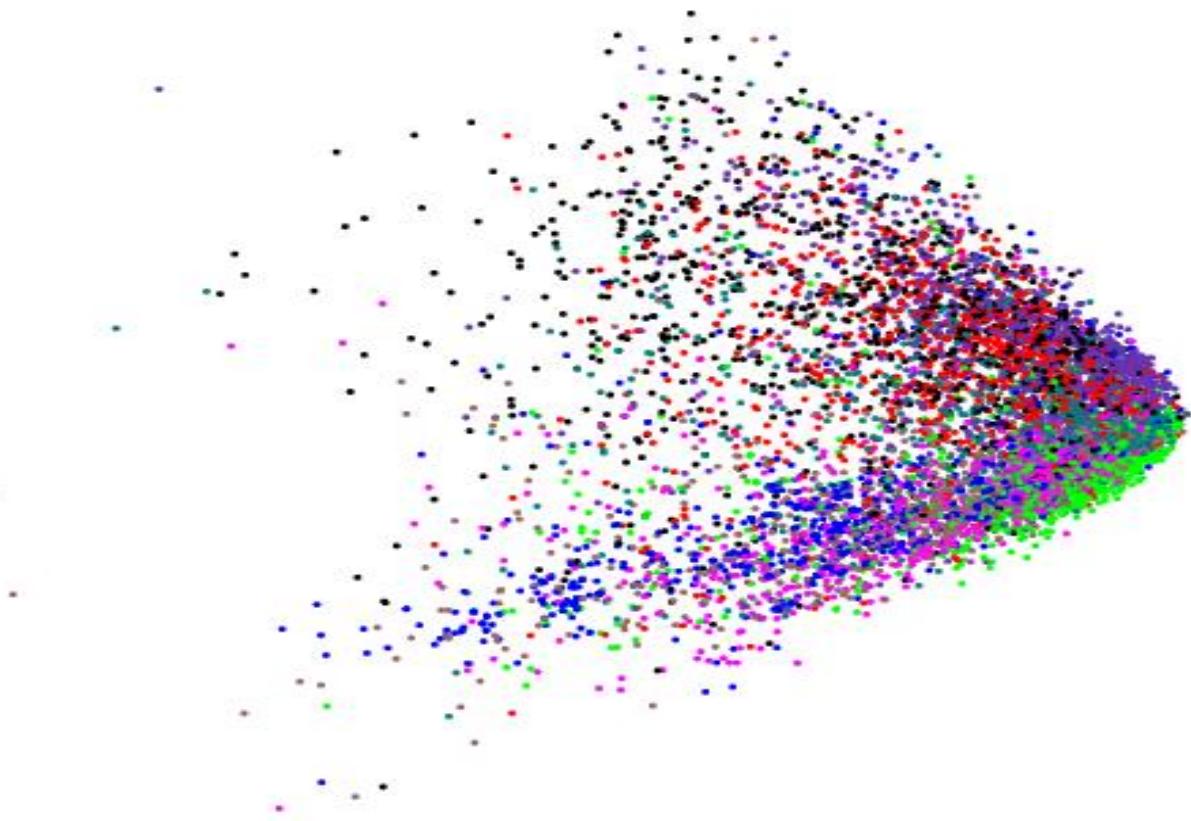
Performance of the autoencoder at document retrieval

- Train on bags of 2000 words for 400,000 training cases of business documents.
 - First train a stack of RBM's. Then fine-tune with backprop.
- Test on a separate 400,000 documents.
 - Pick one test document as a query. Rank order all the other test documents by using the cosine of the angle between codes.
 - Repeat this using each of the 400,000 test documents as the query (requires 0.16 trillion comparisons).
- Plot the number of retrieved documents against the proportion that are in the same hand-labeled class as the query document. Compare with LSA (a version of PCA).

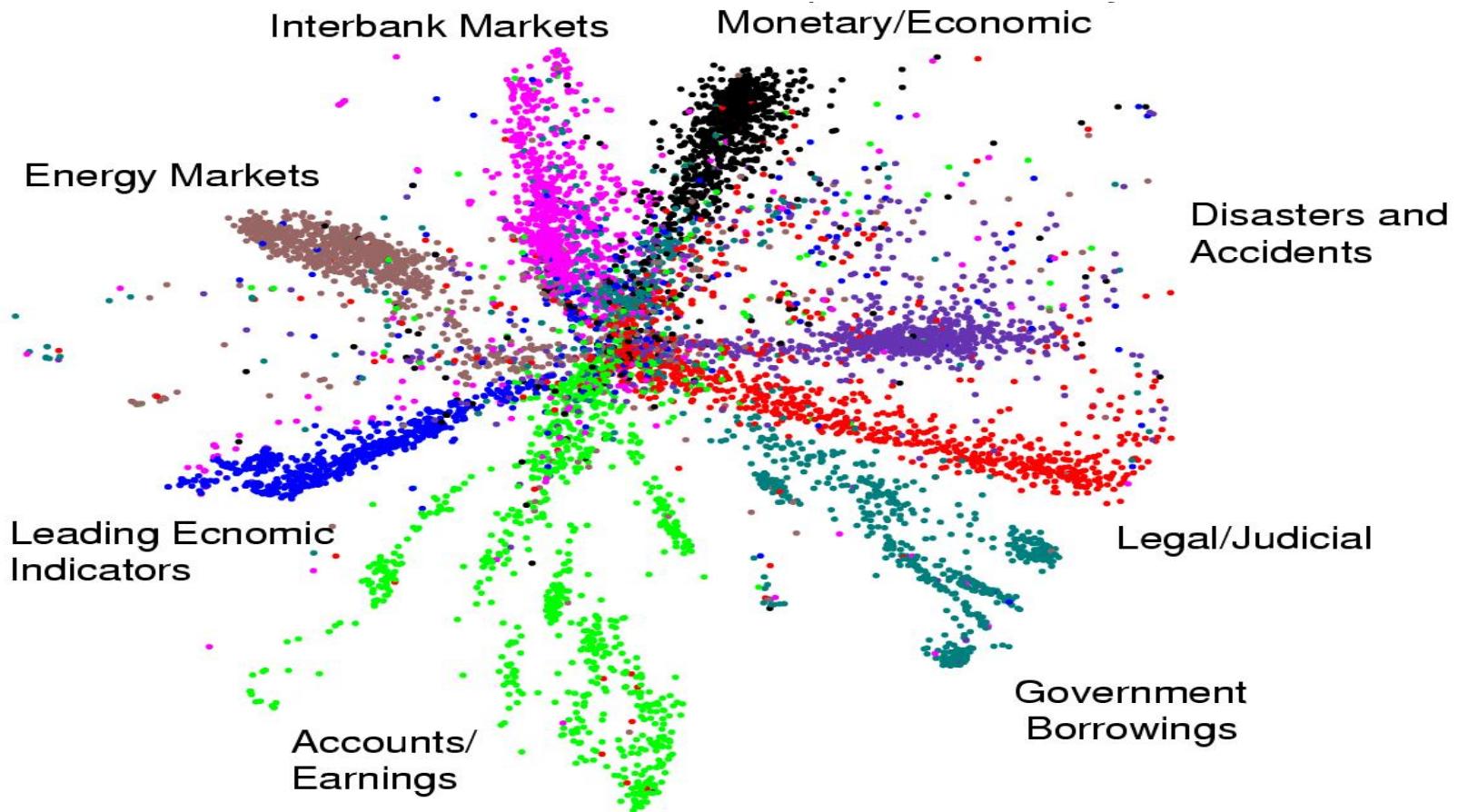
Retrieval performance on 400,000 Reuters business news stories



First compress all documents to 2 numbers using PCA on $\log(1+\text{count})$. Then use different colors for different categories.



First compress all documents to 2 numbers using deep auto.
Then use different colors for different document categories



Neural Networks for Machine Learning

Lecture 15d Semantic hashing

Geoffrey Hinton

Nitish Srivastava,

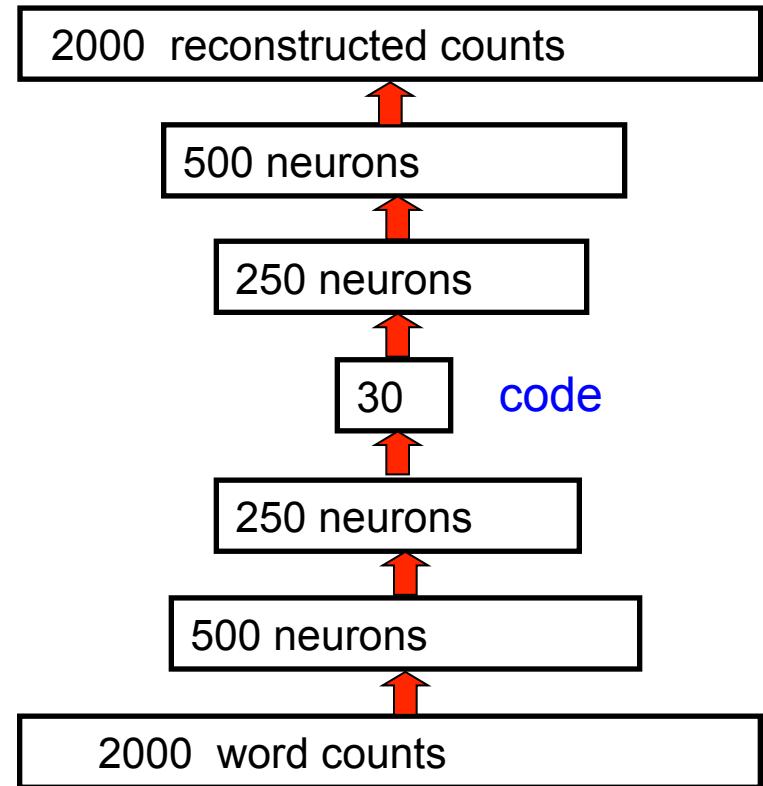
Kevin Swersky

Tijmen Tieleman

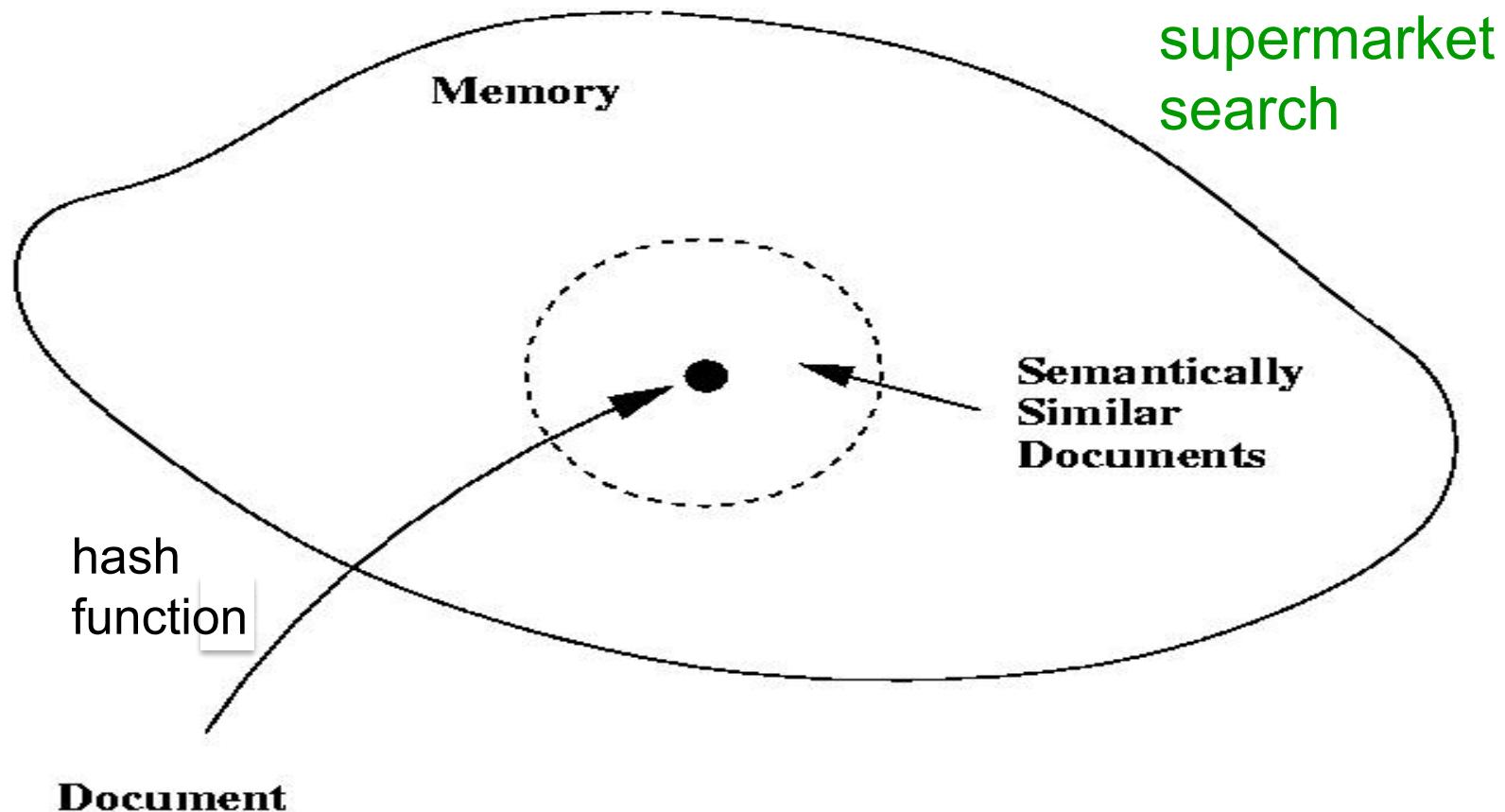
Abdel-rahman Mohamed

Finding binary codes for documents

- Train an auto-encoder using 30 logistic units for the code layer.
- During the fine-tuning stage, add noise to the inputs to the code units.
 - The noise forces their activities to become bimodal in order to resist the effects of the noise.
 - Then we simply threshold the activities of the 30 code units to get a binary code.
- Krizhevsky discovered later that it's easier to just use binary stochastic units in the code layer during training.



Using a deep autoencoder as a hash-function for finding approximate matches



Another view of semantic hashing

- Fast retrieval methods typically work by intersecting stored lists that are associated with cues extracted from the query.
- Computers have special hardware that can intersect 32 very long lists in one instruction.
 - Each bit in a 32-bit binary code specifies a list of half the addresses in the memory.
- Semantic hashing uses machine learning to map the retrieval problem onto the type of list intersection the computer is good at.

Neural Networks for Machine Learning

Lecture 15e

Learning binary codes for image retrieval

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

Binary codes for image retrieval

- Image retrieval is typically done by using the captions. Why not use the images too?
 - Pixels are not like words: individual pixels do not tell us much about the content.
 - Extracting object classes from images is hard (this is out of date!)
- Maybe we should extract a real-valued vector that has information about the content?
 - Matching real-valued vectors in a big database is slow and requires a lot of storage.
- Short binary codes are very easy to store and match.

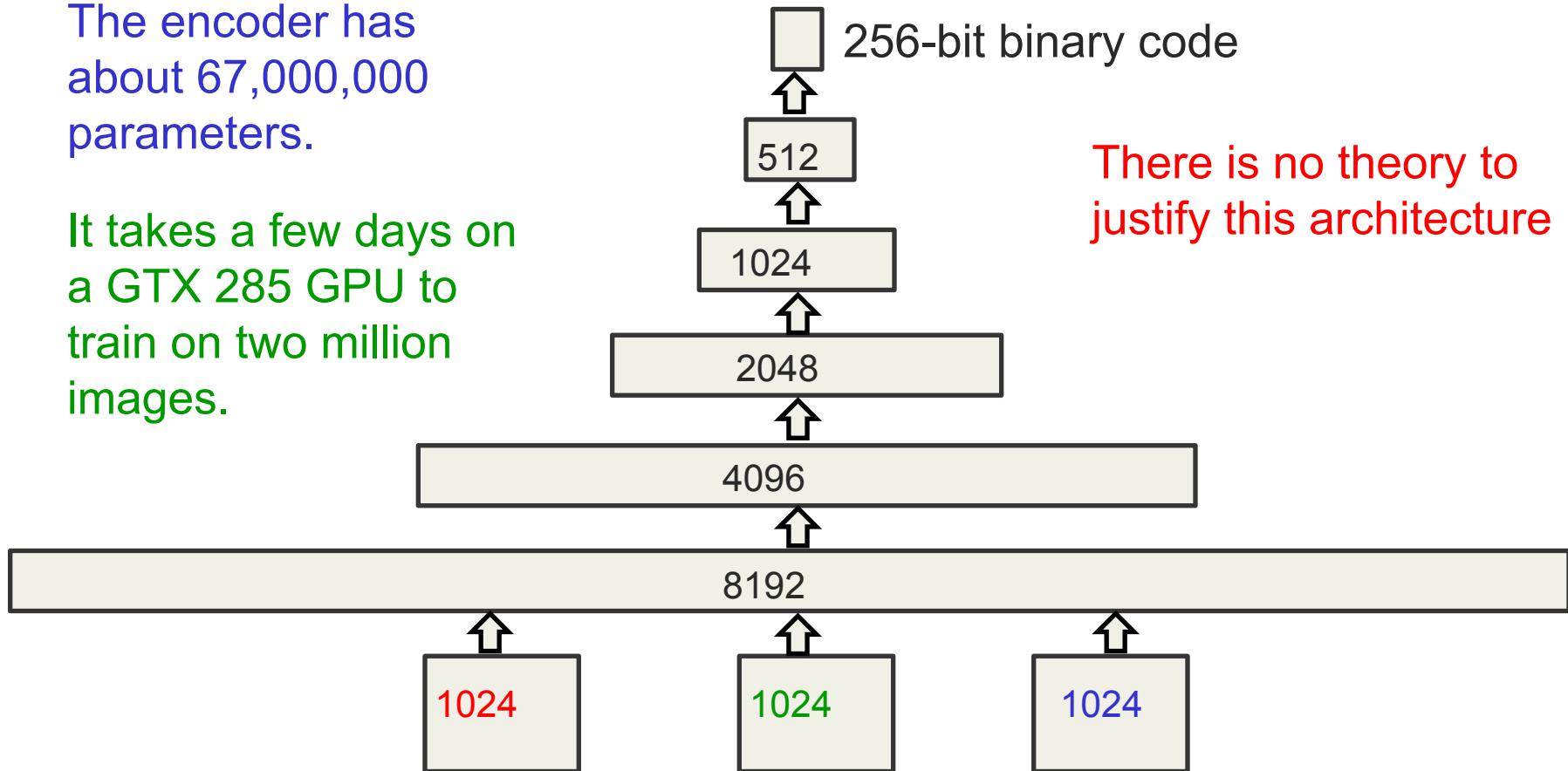
A two-stage method

- First, use semantic hashing with 28-bit binary codes to get a long “shortlist” of promising images.
- Then use 256-bit binary codes to do a serial search for good matches.
 - This only requires a few words of storage per image and the serial search can be done using fast bit-operations.
- But how good are the 256-bit binary codes?
 - Do they find images that we think are similar?

Krizhevsky's deep autoencoder

The encoder has about 67,000,000 parameters.

It takes a few days on a GTX 285 GPU to train on two million images.



Reconstructions of 32x32 color images from 256-bit codes



retrieved using 256 bit codes



retrieved using Euclidean distance in pixel intensity space



retrieved using 256 bit codes

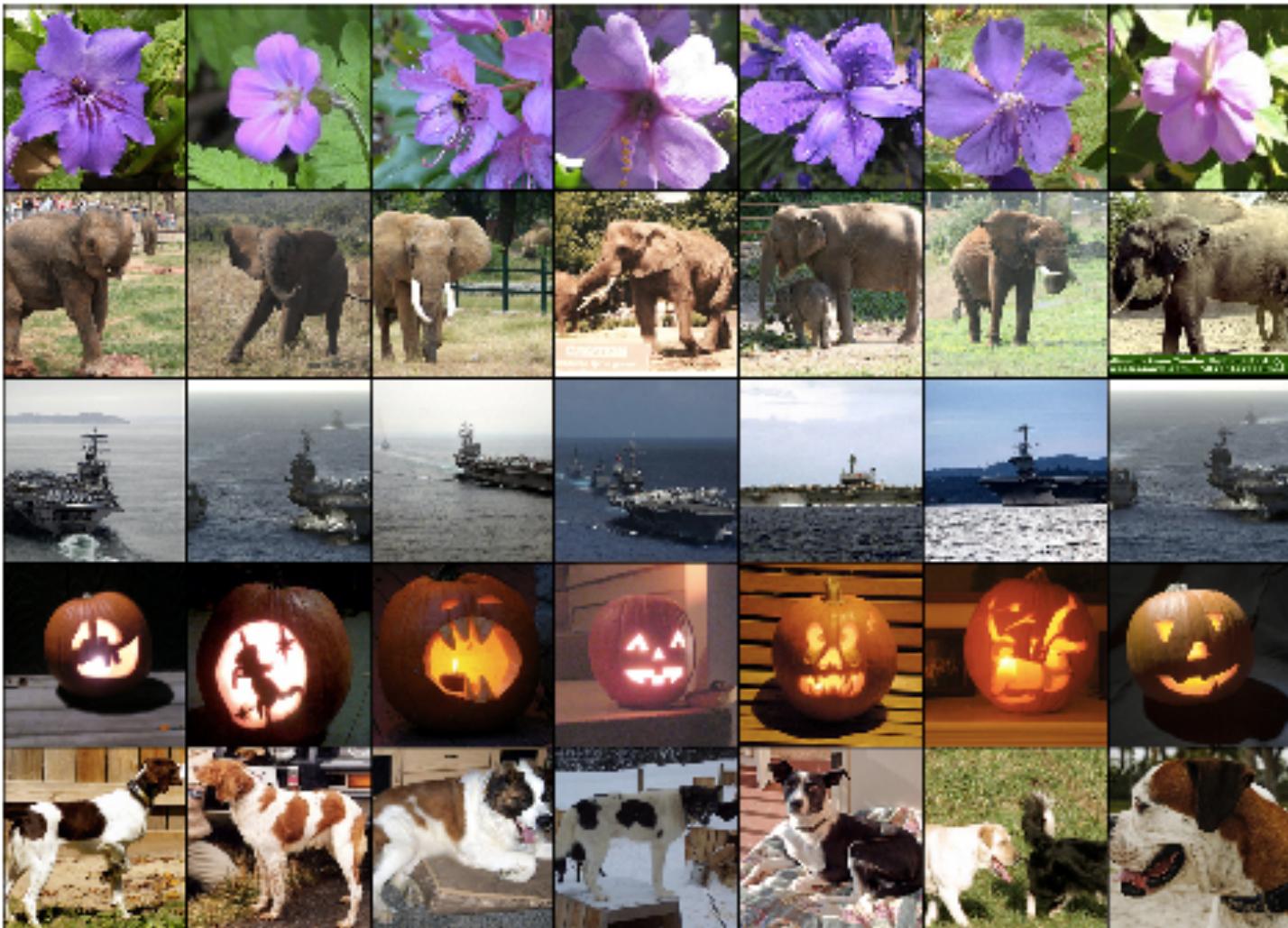


retrieved using Euclidean distance in pixel intensity space



How to make image retrieval more sensitive to objects and less sensitive to pixels

- First train a big net to recognize lots of different types of object in real images.
 - We saw how to do that in lecture 5.
- Then use the activity vector in the last hidden layer as the representation of the image.
 - This should be a much better representation to match than the pixel intensities.
- To see if this approach is likely to work, we can use the net described in lecture 5 that won the ImageNet competition.
- So far we have only tried using the Euclidian distance between the activity vectors in the last hidden layer.
 - It works really well!
 - Will it work with binary codes?



Leftmost column
is the search
image.

Other columns
are the images
that have the
most similar
feature activities
in the last hidden
layer.

Neural Networks for Machine Learning

Lecture 15f Shallow autoencoders for pre-training

Geoffrey Hinton
Nitish Srivastava,
Kevin Swersky
Tijmen Tieleman
Abdel-rahman Mohamed

RBM's as autoencoders

- When we train an RBM with one-step contrastive divergence, it tries to make the reconstructions look like data.
 - It's like an autoencoder, but it's strongly regularized by using binary activities in the hidden layer.
- When trained with maximum likelihood, RBMs are not like autoencoders.
- Maybe we can replace the stack of RBM's used for pre-training by a stack of shallow autoencoders?
 - Pre-training is not as effective (for subsequent discrimination) if the shallow autoencoders are regularized by penalizing the squared weights.

Denoising autoencoders (Vincent et. al. 2008)

- Denoising autoencoders add noise to the input vector by setting many of its components to zero (like dropout, but for inputs).
 - They are still required to reconstruct these components so they must extract features that capture correlations between inputs.
- Pre-training is very effective if we use a stack of denoising autoencoders.
 - It's as good as or better than pre-training with RBMs.
 - It's also simpler to evaluate the pre-training because we can easily compute the value of the objective function.
 - It lacks the nice variational bound we get with RBMs, but this is only of theoretical interest.

Contractive autoencoders (Rifai et. al. 2011)

- Another way to regularize an autoencoder is to try to make the activities of the hidden units as insensitive as possible to the inputs.
 - But they cannot just ignore the inputs because they must reconstruct them.
- We achieve this by penalizing the squared gradient of each hidden activity w.r.t. the inputs.
- Contractive autoencoders work very well for pre-training.
 - The codes tend to have the property that only a small subset of the hidden units are sensitive to changes in the input.
 - But for different parts of the input space, it's a different subset. The active set is sparse.
 - RBMs behave similarly.

Conclusions about pre-training

- There are now many different ways to do layer-by-layer pre-training of features.
 - For datasets that do not have huge numbers of labeled cases, pre-training helps subsequent discriminative learning.
 - Especially if there is extra data that is unlabeled but can be used for pretraining.
 - For very large, labeled datasets, initializing the weights used in supervised learning by using unsupervised pre-training is not necessary, even for deep nets.
 - Pre-training was the first good way to initialize the weights for deep nets, but now there are other ways.
 - But if we make the nets much larger we will need pre-training again!

Neural Networks for Machine Learning

Lecture 16a

Learning a joint model of images and captions

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

Modeling the joint density of images and captions (Srivastava and Salakhutdinov, NIPS 2012)

- Goal: To build a joint density model of captions and standard computer vision feature vectors extracted from real photographs.
 - This needs a lot more computation than building a joint density model of labels and digit images!
- 1. Train a multilayer model of images.
- 2. Train a separate multilayer model of word-count vectors.
- 3. Then add a new top layer that is connected to the top layers of both individual models.
 - Use further joint training of the whole system to allow each modality to improve the earlier layers of the other modality.

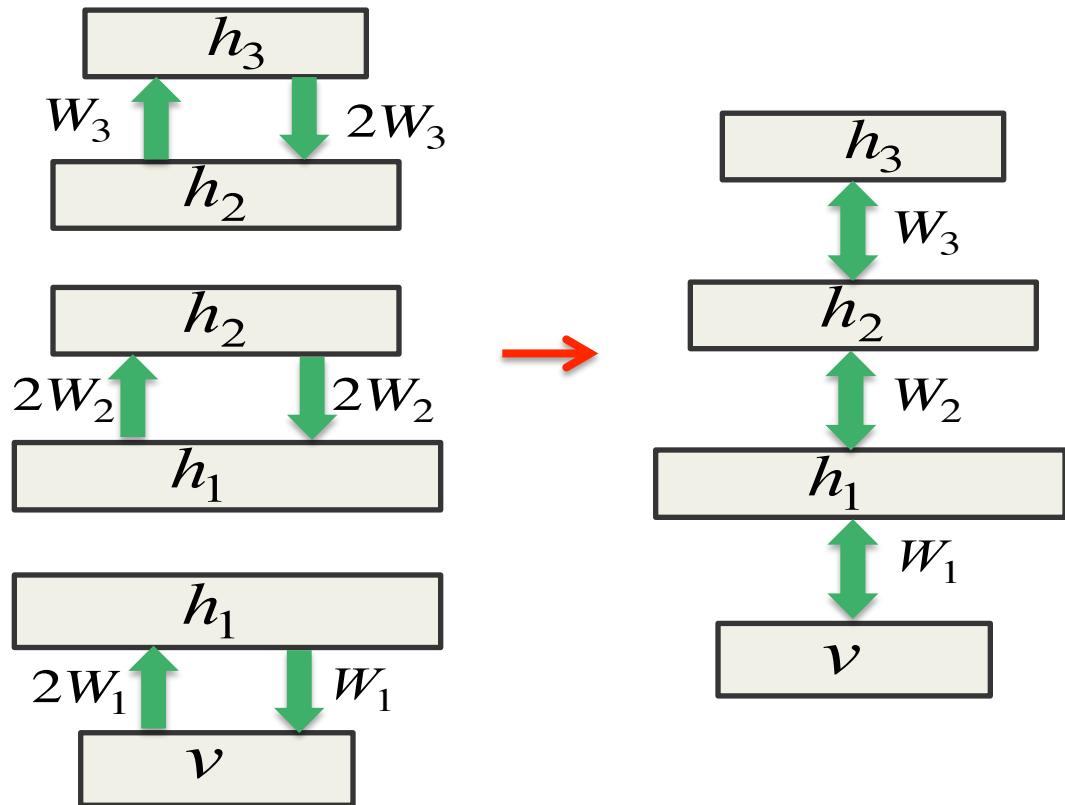
Modeling the joint density of images and captions

(Srivastava and Salakhutdinov, NIPS 2012)

- Instead of using a deep belief net, use a deep Boltzmann machine that has symmetric connections between all pairs of layers.
 - Further joint training of the whole DBM allows each modality to improve the earlier layers of the other modality.
 - That's why they used a DBM.
 - They could also have used a DBN and done generative fine-tuning with contrastive wake-sleep.
- But how did they pre-train the hidden layers of a deep Boltzmann Machine?
 - Standard pre-training leads to composite model that is a DBN not a DBM.

Combining three RBMs to make a DBM

- The top and bottom RBMs must be pre-trained with the weights in one direction twice as big as in the other direction.
 - This can be justified!
- The middle layers do geometric model averaging.



Neural Networks for Machine Learning

Lecture 16b Hierarchical coordinate frames

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

Why convolutional neural networks are doomed

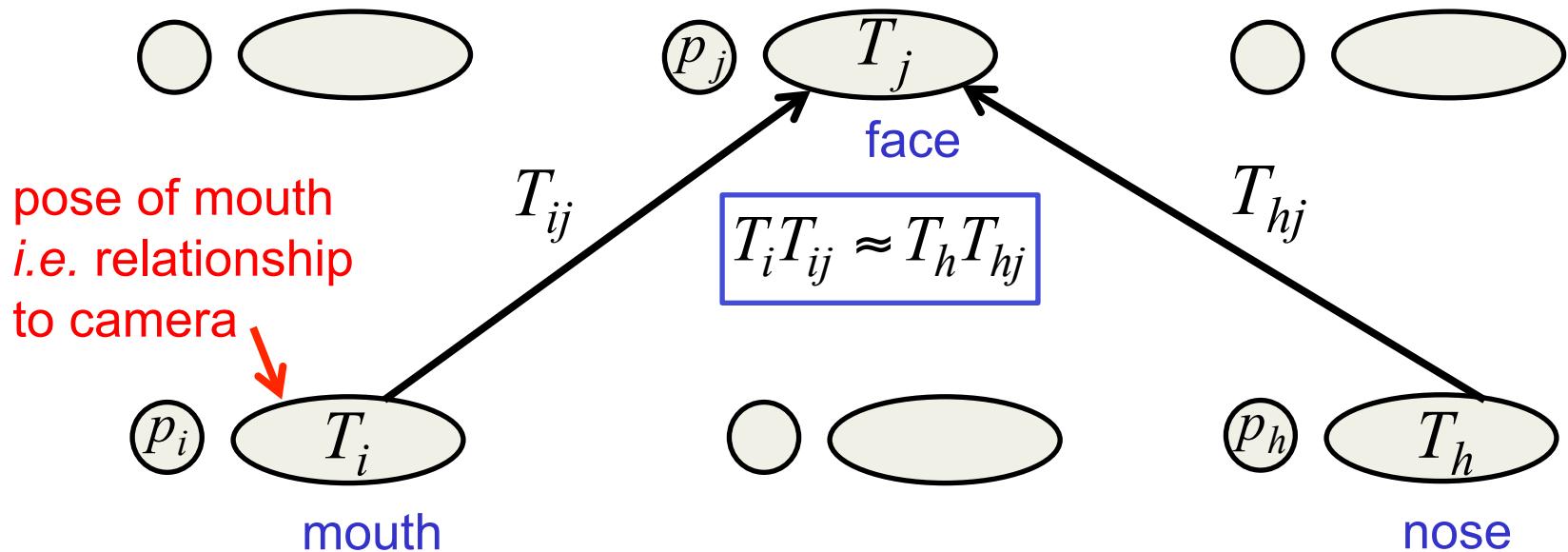
- Pooling loses the precise spatial relationships between higher-level parts such as a nose and a mouth.
 - The precise spatial relationships are needed for identity recognition.
 - Overlapping the pools helps a bit.
- Convolutional nets that just use translations cannot extrapolate their understanding of geometric relationships to radically new viewpoints.
 - People are very good at extrapolating. After seeing a new shape once they can recognize it from a different viewpoint.

The hierarchical coordinate frame approach

- Use a group of neurons to represent the conjunction of the shape of a feature and its pose relative to the retina.
 - The pose relative to the retina is the relationship between the coordinate frame of the retina and the intrinsic coordinate frame of the feature.
 - Recognize larger features by using the consistency of the poses of their parts.
- 
- nose and mouth make consistent predictions for pose of face
- nose and mouth make inconsistent predictions for pose of face

Two layers in a hierarchy of parts

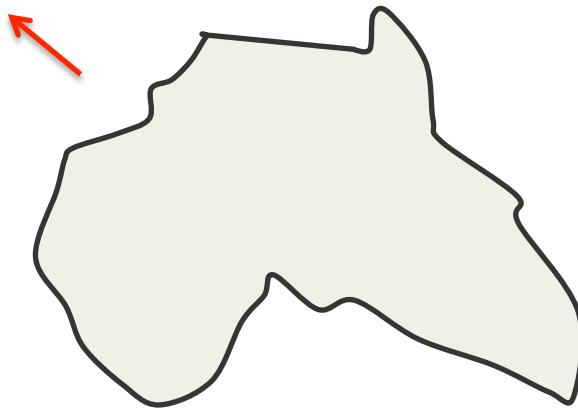
- A higher level visual entity is present if several lower level visual entities can agree on their predictions for its pose (**inverse computer graphics!**)



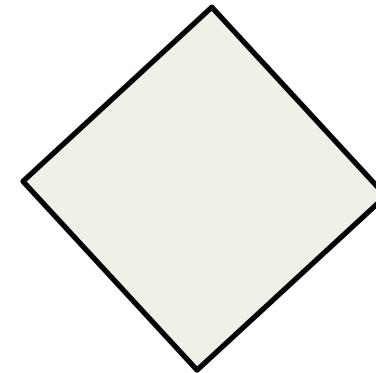
A crucial property of the pose vectors

- They allow spatial transformations to be modeled by linear operations.
 - This makes it easy to learn a hierarchy of visual entities.
 - It makes it easy to generalize across viewpoints.
- The invariant geometric properties of a shape are in the weights, not in the activities.
 - The activities are equivariant: As the pose of the object varies, the activities all vary.
 - The percept of an object changes as the viewpoint changes.

Evidence that our visual systems impose coordinate frames in order to represent shapes (after Irvin Rock)



What country is this? Hint: Sarah Palin



The square and the diamond are very different percepts that make different properties obvious.

Neural Networks for Machine Learning

Lecture 16c

Bayesian optimization of neural network hyperparameters

Geoffrey Hinton

Nitish Srivastava,

Kevin Swersky

Tijmen Tieleman

Abdel-rahman Mohamed

Let machine learning figure out the hyper-parameters!

(Snoek, Larochelle & Adams, NIPS 2012)

- One of the commonest reasons for not using neural networks is that it requires a lot of skill to set hyper-parameters.
 - Number of layers
 - Number of units per layer
 - Type of unit
 - Weight penalty
 - Learning rate
 - Momentum *etc. etc.*
- Naive grid search: Make a list of alternative values for each hyper-parameter and then try all possible combinations.
 - Can we do better than this?
- Sampling random combinations:
This is much better if some hyper-parameters have no effect.
 - Its a big waste to exactly repeat the settings of the other hyper-parameters.

Machine learning to the rescue

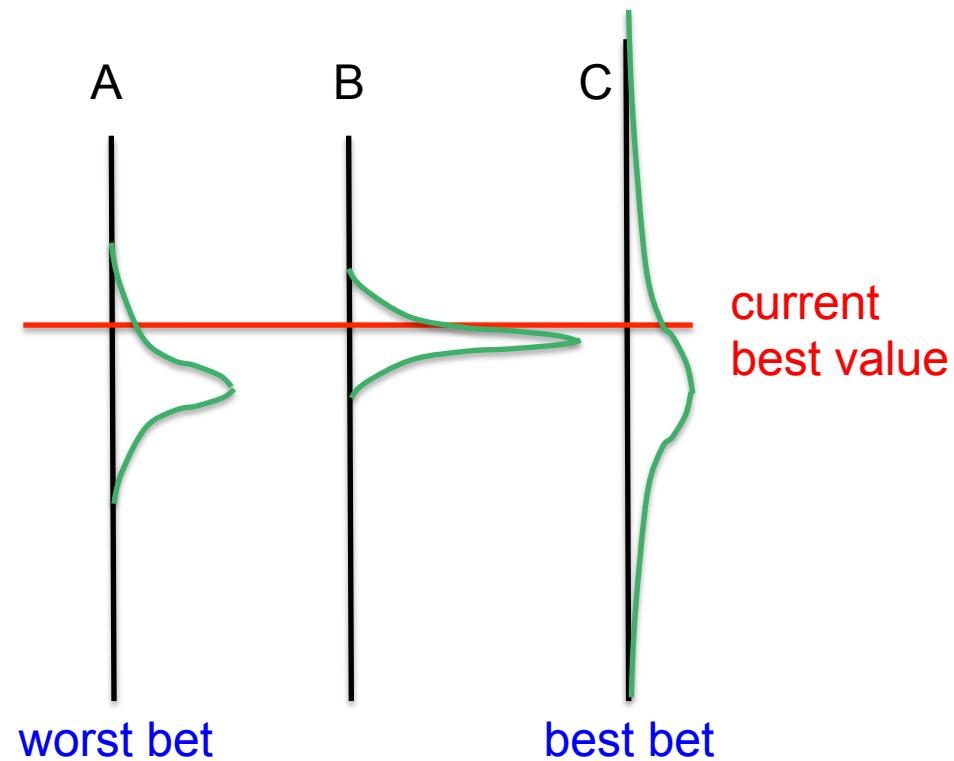
- Instead of using random combinations of values for the hyper-parameters, why not look at the results so far?
 - Predict regions of the hyper-parameter space that might give better results.
 - We need to predict how well a new combination will do and also model the uncertainty of that prediction.
- We assume that the amount of computation involved in evaluating one setting of the hyper-parameters is huge.
 - Much more than the work involved in building a model that predicts the result from knowing previous results with different settings of the hyper-parameters.

Gaussian Process models

- These models assume that similar inputs give similar outputs.
 - This is a very weak but very sensible prior for the effects of hyper-parameters.
- For each input dimension, they learn the appropriate scale for measuring similarity.
 - Is 200 similar to 300?
 - Look to see if they give similar results in the data so far.
- GP models do more than just predicting a single value.
 - They predict a Gaussian distribution of values.
- For test cases that are close to several, consistent training cases the predictions are fairly sharp.
- For test cases far from any training cases, the predictions have high variance.

A sensible way to decide what to try

- Keep track of the best setting so far.
- After each experiment this might stay the same or it might improve if the latest result is the best.
- Pick a setting of the hyper-parameters such that the **expected improvement** in our best setting is big.
 - don't worry about the downside (hedge funds!)



How well does Bayesian optimization work?

- If you have the resources to run a lot of experiments, Bayesian optimization is much better than a person at finding good combinations of hyper-parameters.
 - This is not the kind of task we are good at.
 - We cannot keep in mind the results of 50 different experiments and see what they predict.
- It's much less prone to doing a good job for the method we like and a bad job for the method we are comparing with.
 - People cannot help doing this. They try much harder for their own method because they know it ought to work better!

Neural Networks for Machine Learning

Lecture 16d The fog of progress

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

Why we cannot predict the long-term future

- Consider driving at night. The number of photons you receive from the tail-lights of the car in front falls off as $1 / d^2$
- Now suppose there is fog.
 - For small distances its still $1 / d^2$
 - But for big distances its $\exp(-d)$ because fog absorbs a certain fraction of the photons per unit distance.
- So the car in front becomes completely invisible at a distance at which our short-range $1 / d^2$ model predicts it will be very visible.
 - This kills people.

The effect of exponential progress

- Over the short term, things change slowly and its easy to predict progress.
 - We can all make quite good guesses about what will be in the iPhone 6.
- But in the longer run our perception of the future hits a wall, just like fog.
- So the long term future of machine learning and neural nets is a total mystery.
 - But over the next five years, its highly probable that big, deep neural networks will do amazing things.