University of Toronto

Neural Networks for Machine Learning

Geoffrey Hinton - Summer 2015

Contributors: Max Smith

Latest revision: August 13, 2015

Contents

Abstract

Todo

1 Introduction

2 The Perceptron learning procedure

3 The backpropagation learning proccedure

4 Learning feature vectors for words

5 Object recognition with neural nets

5.1 Why object recognition is difficult

- Things that make it hard to recognize objects:
  - **Segmentation:** Real scenes are cluttered with other objects
    - It's 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:
    - eg. a hand-written 2 can have a large loop or just a cusp
  - **Affordances:** Object classes are often defined by how the are used
  - **Viewpoint:** Changes in viewpoint cause changes in images that standard learning methods cannot cope with

5.2 Achieving viewpoint invariance

- Several approaches:
  - Use redundant invariant features
  - Put a box around the object and use normalized pixels
  - Use replciated features with pooling. This is called "convolution neural nets"
  - Use a hierarchy of parts that have explicit poses relative to the camera

The invariant feature approach

- Extract a large, redudant set of features that are invariant under transformations
- With enough invariant features, there is only one way to assemble them into an object
- 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 normlized 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, etc.
- Chooosing the box is difficult because of: segmetnation erros, occlusion, unusual orientations
- We need to recognize the shape to get the box right (chicken and egg dilema)

The brute force normalization approach

- When training the recognizer, use well-segmented, upright images to fit the correct box
- At test time try all possble boxes in a range of positiosn 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

6 Optimization: How to make the learning go faster

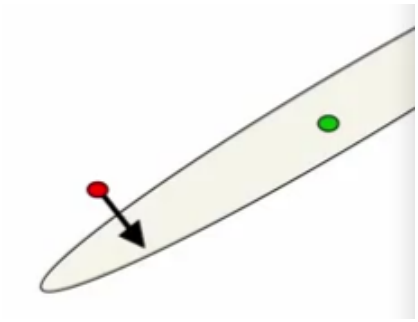
Overview of mini-batch gradient descent

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
- For multi-layer, non-linear nets the error surface is much more complicated
  - But locally, a piece of a quadratic bowl is an approximation

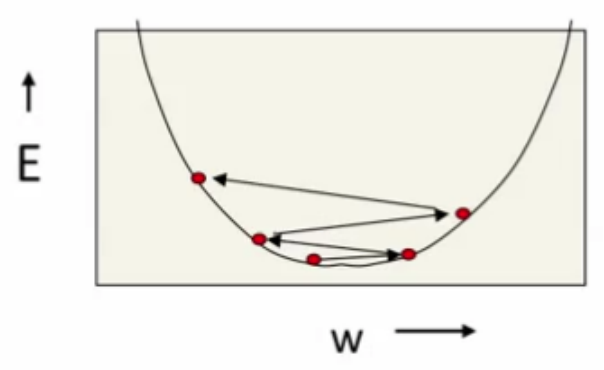
Convergence speed of full batch learning when the error surface is a quadratic bowl

- 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 (larger axis of ellipse, where we only want to travel a small distance)
- The gradient is small in the direction in which we want to travel a large distance (towards the center on the smaller axis).
- 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 acoss the ravine (this can diverge)



- What we want:
  - 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
- An extreme version of this approach updates weights after each case. It's called "online."
- Mini-batches are usually better than online (10, 100, 1000)
  - 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
- Ideal would be exactly same number of each class type in each mini-batch
- Avoid mini-batches that are very uncharacterstic of the data (e.g., all the same class)

Two types of learning algorith

- 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

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 autmate this way of adjusting the learning rate
- Towards the end of mini-batch learning it always nearly 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

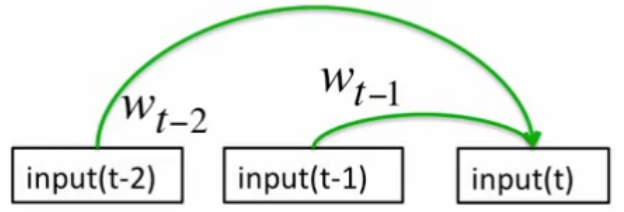
7 Recurrent neural networks

7.1 Modeling sequences: A brief overview

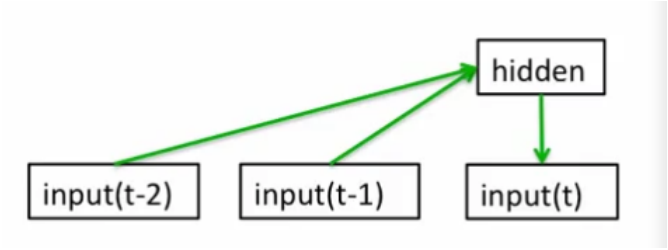
- When applying machine learning to sequences, we often want to turn an input sequence into an output sequence that lives in a different domain
  - e.g., turn a sequene of sound pressures into words
- When there is no separate target sequence, we can get a teaching signal by trying to predict the next term in the input sequence
  - The target ouput sequence is the input sequence with and 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 (unsupervised)

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**: generalize autoregressive models by using one or more layers of non-linear hidden units (e.g., Bengio's first language modle)

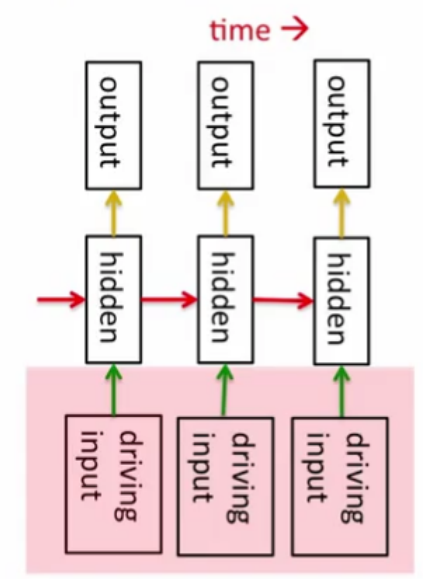


Beyond memoryless models

- 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

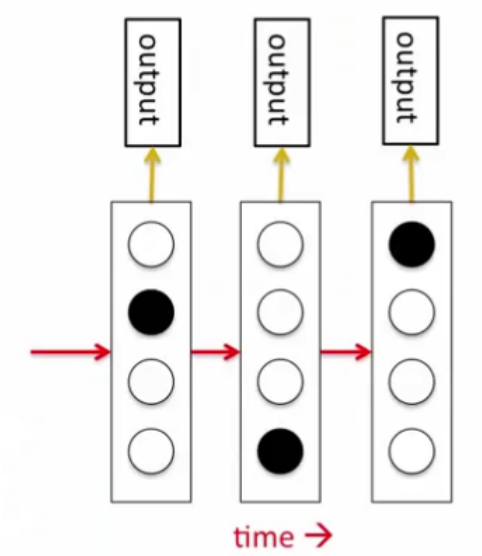
Linear Dynamical Systems

- These are generative models. They have a real-valued hidden state that cannot be observed directly.
  - The hidden state has linear dynamics with Guassian noise and produces the observations using a linear model with Gaussian noise.
  - There may also be driving inputs
- To predict the next ouput (so that we can shoot down the missile) we need to infer the hidden state



Hidden Markov Models

- HMM 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

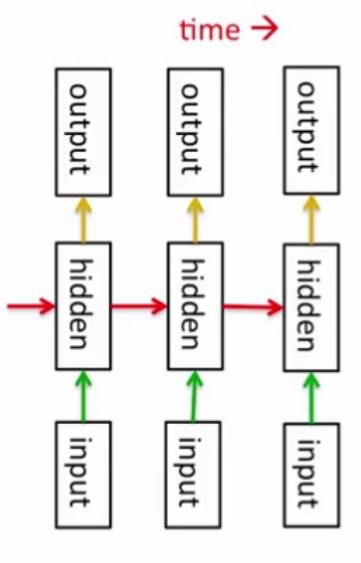
- To predict the next ouput we need to infer the probability distribution over hidden states
- HMMs have efficient algorithms for inference and learning
- Fundamental limitation: consider what happens when a HMM generates data

  - 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)
  - The semantics and intonation needs to fit
  - The accent, rate, volume, and vocal tract characteristics must all fit
- All of these aspects combined could be 100 bits of information that's huge!

Recurrent neural networks

- RNNs are very powerufl, 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

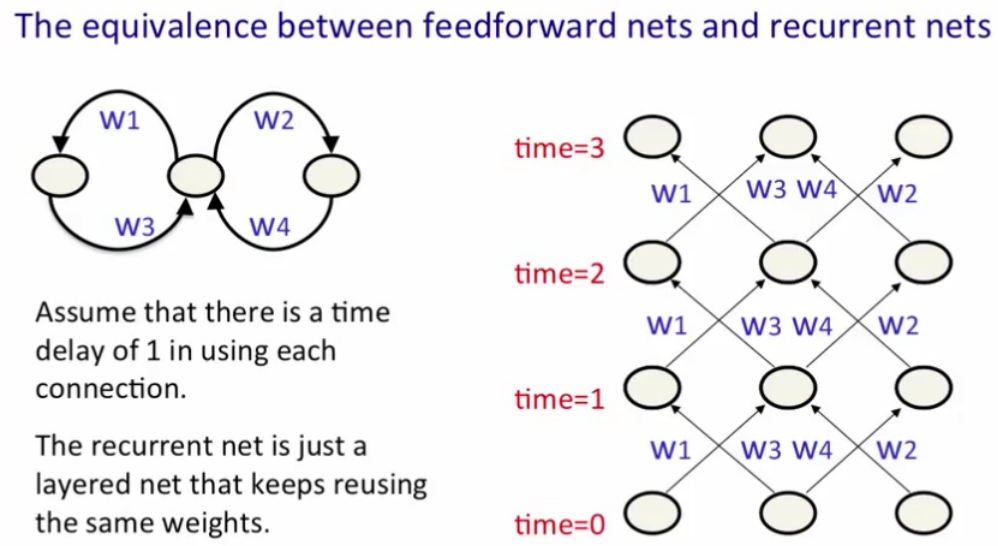


- With enough neurons and time, RNNs can compute anything that can be computed by your computer
- WHat kinds of behaviour can RNNs exhibit?
  - They can oscillate (good for motor control)
  - They can settle to point attractors (retrieving memories)
  - Behave chaotically (bad for information processing)
  - 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
- Computational power of RNNS make them very had to train

- **Do generative models need to be stochastic?**
- LDS & HMM are stochastic
  - The posterior probability distribution over the hidden states given the observed data so far is a deterministic function of the data
- RNN are deterministic
  - So think fo the hidden state of an RNN as the equivalent of the deterministic probability distribution over hidden states in a LDS or HMM

7.2 Training RNNs with backpropagation

- The recurrent net is just a layered net that keeps reusing the same weights.
- Layered feedforward network, where the weights are constrained to be the same at each layer



Reminder: Backpropagation with weight constraints

- It is easy to modify the backprop alg to incorporate linear constraints between the weights
- We compute the gradients as usual, and then modify the gradients so that they satsify the constraints
  - To constrain: $w_1 = w_2$
  - We need: $\delta w_1 = \delta w_2$
  - compute: $\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}$
  - use $\frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2}$ for w_1 and w_2
- So if the weights started off satisfying constraints, they will continue too

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
- Training algorithm in the time domain
  - Forward pass builds up activities at each time step
  - Backward pass peels activities off stack and computes error derivatives at each time step
  - Add together the derivatives at all the different times for each weight
- We need to specify the initial activity state of all the hidden and ouput units
  - We could 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 we learn the weights
    - Start off with an initial guess for the initial states
    - At the end of each training sequence, backpropagate through time all the way to the intial states to get the gradient of the error function with respect to each inital state
    - Adjust initial states by negative gradient

Providing input to recurrent networks

- Several options:
  - Specify the input states of all the units
  - Specify the initial states of a subset of units
  - Specify the states of the same subset of the units at every time step (natural way to model most sequential data; e.g., of the previous figure the first column of neurons)
- We can specify targets in several ways too:
  - Desired final activities of all the units
  -

8 More recurrent neural networks

9 Ways to make neural networks generalize better

10 Combining multiple neural networks to improve generalization

11 Hopfield nets and Boltzmann machines

12 Restricted Boltzmann machines (RBMs)

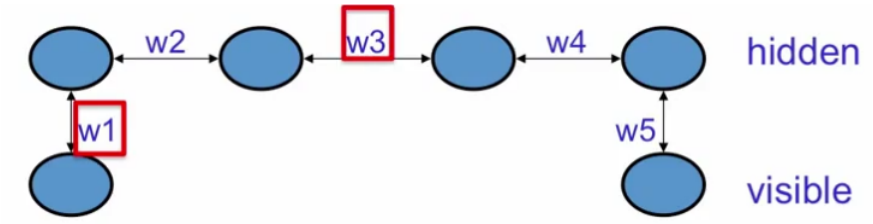
12.1 Boltzmann machine learning

- Unsupervised learning problem

- We want the maximize the product of the probabilities that the Boltzmann machine assigns to the binary vectors in the training set
- Equivalent to maximizing the sum of the log probabilities that the Boltzmann machine assigns to the training vectors
- It is also equivalent to maximizing hte 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 the chain of units:



- If the raining 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 w1 or w5 we must know w3)
- Everything that one weight needs to know about the other weights and the data is contained in the difference of two correlations:

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

Derivative of log probability of one training vector, v under the model.

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

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

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

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

Why is the derivative so simple?

- THe probability of a global configuration at thermal equilibrium is an exponential function of its energy
- So setting to equilibrium makes the log probability a linear function of the energy

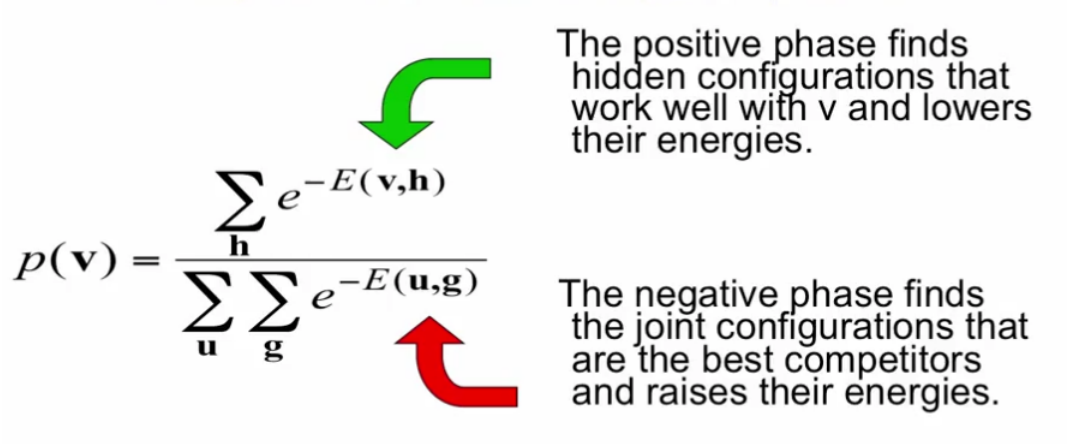
- The energy is a linear fnuction of the weights and states, so:

$$-\frac{\partial E}{\partial w_{ij}} = s_i s_j$$

- The process of settling to thermal equilibrium propagates information about the weights (We dontt need backprop)

Why do we need the negative phase?

- Probability of a visible vector:



An inefficient way to collect the statistics required for learning

- **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
- Once equilibrium, Sample $s_i s_j$ for every connected pair of units
- Repeat for al ldata vectors in the training set and average
- **Negative phase**: Set all the units to random states
- Update until equilibrium at temperature of 1
- Sample $s_i s_j$ for every connected pair of units
- Repeat many times and average to get good estimates

13 Stacking RBMs to make Deep Belief Nets

14 Deep neural nets with generative pre-training

15 Modeling hierarchical structure with neural nets

16 Recent applications of deep neural nets