# Notes for Geoff Hinton's Coursera ML course

fcrimins edited this page on 10 Jul · 261 revisions

Octave source: https://ftp.gnu.org/gnu/octave/

Also see accompanying green Staples notebook.

Also see the notes in lecture slides in
file:///home/fred/Documents/articles/geoff_hinton's_machine_learning_coursera/

Also see this other set of lecture notes:
http://www.cs.toronto.edu/~tijmen/csc321/lecture_notes.shtml

Machine Learning Notes

Hinton's Advanced Machine Learning

## Week 2: Types of Neural Network Architectures

- Ilya Sutskever (2011) trained special type or RNN to predict next char in sequence
  - it generates text by predicting the probability distribution for the next char, not the highest likely next char (which would generate text like "the united states of the united states of the..."), and then sampling from that distribution
- symmetric nets are much easier to analyze than RNNs (John Hopfield)
  - but they're more restricted, e.g. can't learn cycles

## Week 2: What perceptrons can't do

- A perceptron cannot recognize patterns under translation if we allow wraparound.
  - E.g. 2 binary input vectors, A and B, each with 4 out of 10 activated "pixels." Each of the 10 pixels will be activated by 4 translations of A and of B, so the total input received by the decision unit over all these patterns, in both cases, 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.
  - However, if we have 3 patterns assigned to two classes, A and B, and A contains a pattern with 4 pixels while B contains patterns with 1 and 3 pixels then a *binary* decision unit can classify if we allow translations and wraparound. Weight = 1 from each pixel with bias of -3.5
  - Minskey and Papert's "Group Invariance Theorem" says that the part of a perceptron that learns cannot do this if the transformations form a group (e.g. translations with wraparound).
  - This result is devastating for pattern recognition (PR) because the whole point of PR is to recognize patterns that undergo transformations, like translation.
  - 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 PR must be solved by the hand-coded feature detectors, not the learning procedure.
  - Networks without hidden units are very limited in what they can learn to model.

## Week 3: Learning the weights of a linear neuron

- Instead of showing the weights get closer to a good set of weights (i.e. perceptrons, which suffer from 2 good set of weights do not average to a good set) show that actual output values get closer to the target values.
  - In perceptron learning the outputs can get farther away from the targets, even though the weights are getting closer.

**Pages** 70

Find a Page…

**Home**

**A Few Notes on Systematic Trading**

**AdaBoost Notes**

**Agile Notes**

**Apache Spark**

**Assorted Ideas**

**Better Java**

**Big Data Architecture**

**Computer Vision**

**Data**

**Data Mining**

**Data Science**

**Databricks Notes**

**Differential Gene Expression**

**Docker Notes**

Show 55 more pages…

**Clone this wiki locally**

https://github.com/fcrimi

- The "delta rule" for learning: Δw_i = epsilon * x_i * (t - y) ... where epsilon := "learning rate", t := target/true output, and y := estimated output
  - Derivation: Error = 0.5 * Sum_{n in training}[(t_n - y_n)^2] ... the 1/2 is only there to make the 2 in the derivative cancel
  - Differentiate the error wrt one of the weights, w_i: ∂E/∂w_i = 0.5 * Sum_n[∂y_n/∂w_i * ∂E_n/∂y_n] ... Chain Rule ("easy to remember, just cancel those 2 ∂y_ns ... but only when there aren't any mathematicians looking) ... = -Sum_n[x_{i,n} * (t_n - y_n)]
    - ∂y_n/∂w_i = x_{i,n} because y_n = w_i * x_{i,n}
    - dE_n/dy_n is just the derivative of the (squared) Error function
  - Therefore: Δw_i = -epsilon * ∂E/∂w_i

## Week 3: The error surface for a linear neuron

- Difference between "batch" and "on-line"
  - Simplest batch learning does steepest gradient descent
  - On-line/stochastic zig-zags between training case "lines" at each step moving perpendicularly to a line. Imagine the intersection of 2 training case lines and moving perpendicularly back and forth perpendicularly to both while converging on their intersection point.
    - This is very slow if the variables are highly correlated (very elongated ellipse) because the perpendicular updates (the gradients) are also perpendicular to the intersection.
    - FWC idea - look at angle between consecutive gradients to detect correlated dimensions

## Week 3: The backpropagation algorithm

- Networks without hidden layers are very limited in the input-output mappings they can model
- Adding a layer of hand-coded features (as in 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 of different features
  - We need to automate the trial and error feature designing loop
  - FWC - but again the trick here is to properly incorporate priors into this design
- **Reinforcement learning: learning by perturbing weights (an idea that occurs to everyone)**
  - Randomly perturb one weight and see if it improves performance--if so, save the change
  - Very inefficient--requires multiple forward passes on a set of training cases just to update one weight. Backprop much better "by a factor of the number of weights in the network."
  - Could randomly perturb all the weights in parallel and correlate performance gain w/ weight changes, but not much better b/c requires lots of trials on each training case to "see" the effect of changing one weight through the noise created by changing all the others (FWC - reminds me of the Shapley optimization)
  - Better idea: randomly perturb the activities of the hidden units. Once we know how we want a hidden activity to change on a training case, we can compute how to change the weights. There are fewer activities than weights, but backprop still wins by a factor of the number of neurons.
  - Finite Difference Approximation (compute +/- epsilon changes for each weight and move in that direction) works, but backprop finds the exact gradient (**∂E/∂w_{i,j}** = Sum_j[∂z_j/∂w_{i,j} * ∂E/∂z_j] = Sum_j[w_{i,j} * ∂E/∂z_j]) much more quickly.
- FWC - machine learning (backprop) is all about the learning rate! So you can either be smart (and use backprop) or buy more computers. You're only constrained if you need both. Search Google for: "machine learning for the maximization of an arbitrary function"
- *Instead of using pre-set coefficients (desired activities) to train engineered features, use error derivatives wrt hidden activities.* We can compute error derivatives for all of the hidden units efficiently at the same time: Once we have the error derivatives for the hidden activities (hidden neuron output) it's easy to compute the (input) weights going into a hidden unit.

## Week 3: Using the derivatives computed by backprop

- 2 types of noise: unreliable target values (small worry), sampling error (big worry)
- When we fit the model it cannot tell which regularities are real and which are caused by sampling error. FWC - So are there methods then to distinguish between the two (besides e.g. cross validation)??? See week 7.
    - Weight decay - keep weights near 0
    - Weight sharing - keep some weights similar to each other
    - Early stopping - peek at a fake test set while training and stop when performance gets decent
    - Model averaging - train lots of NNs and average then
    - Bayesian fitting - fancy averaging
    - Dropout - randomly omitting hidden units when training
    - Generative pre-training
    - FWC idea - other constraints such as monotonicity and limits on distributions
        - FWC - **monotonicity could be enforced in an auto-encoder with skip connections from the units that are desired to be monotonic straight up to the cost function**

## Week 4: Neural nets for machine learning

- Obvious way to express regularities is as symbolic **rules**
    - but finding the symbolic rules involves a difficult search through a large discreet space
    - so model as a NN instead w/
        - input := person1 + relationship (both 1-hot encodings)
        - output := person2 (also 1-hot)
- **Instead of predicting the 3rd term in a relationship, [A R B], we could provide all 3 as input and predict P([A R B] is correct)**
    - for this we'd need a whole bunch of "correct" facts as well as "incorrect" ones (fwc - **negative sampling**)

## 4b: A brief diversion into cognitive science

[probably not interesting if you're an engineer]

- 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. [FWC - i.e. just "seeing" the answer is the same as just "seeing" which rules 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”. "*A many-to-many mapping between concepts and neurons.*"
    - a "**local**" representation of a word is a 1-hot vector of length equal to the size of the vocab
    - a "**distributed**" representation is a word embedding which is distributed b/c it is based on all the other words

## 4c: The softmax output function

softmax a way of forcing the outputs to sum to 1 so that they can represent a probability distribution across discrete, mutually exclusive alternatives

- 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.
    - [FWC - **I wonder if the changes to the loss function that are about to be described are a *dual* of a representation of constraints as output neurons. E.g. could the effects of this different loss function be obtained by changing the architecture of the output?**]
- The output units in a softmax group use a non-local non-linearity:
    - z_i input to final layer, Sum_i[z_i] != 1
    - then scale "softmax group(s)" so that they sum to 1: y_i = **e^**(z_i) / Sum_j[**e^**(z_j)] ... (**don't forget the e's!**)
    - **∂y_i/∂z_i = y_i * (1 - y_i)** ... not trivial to derive b/c of all the terms in the numerator above
- **Cross-entropy**: the *right* cost function to use with softmax
    - The right cost function is the **negative log probability of the right answer**. [FWC - because the answer is a 1-hot vector with a 1 at the right answer or this from Sebastian Ruder: "have a look at Karpathy's explanation to gain some more intuitions about the connection between softmax and cross-entropy"]
        - C = -Sum_j[t_j ln(y_j)] ... where t_j == 1 for only one j (note the multiplication of t_j and ln(y_j) not subtraction)
        - C = -log( exp(y_i) / Sum_j[exp(y_j)] ) ... where i is the right answer (this is from Quiz 4, question 1) ... = -y_i + log(Sum_j[exp(y_j)])
    - 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 0.000000001 (for a target value of 1)
        - Effectively, the steepness of dC/dy exactly balances the flatness of dy/dz
            - ∂C/∂z_i = Sum_j[ ∂C/∂y_i * ∂y_i/∂z_i ] = **y_i - t_i** .... (the chain rule again)

## Lecture 4d: Neuro-probabilistic language models

- Bengio's NN for predicting the next word (see green Staples notebook or pdfs)
- 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 (lower dimensioned) feature representation also allows for a context that contains many more previous words (e.g. 10).]

## Lecture 4e: Dealing with a large number of possible outputs

- embed words in 2D, 2 approaches (see green Staples notebook or pdfs)
  - NN to predict logit
  - tree-based approach (Minih, Hinton 2009)
- a simpler way to learn feature vectors of words (Collobert, Weston, 2008)
  - introduction to negative sampling
- eventually apply dimension reduction (t-SNE and How to Use t-SNE Efficiently) to display these vectors in a 2D map

## Week 4 Quiz

- (for some reason I can't get numbered lists to work unless they are sublists)
  i. The cross-entropy cost function with an *n*-way softmax unit (a softmax unit with *n* different outputs) is equivalent to: (answer) the cross entropy cost function with n logistic units
  - FWC - reason: b/c softmax is just a scaling of logistic
  - wrong - correct answer "None of the above"

  ii. A 2-way softmax unit (a softmax unit with 2 elements) with the cross entropy cost function is equivalent to: (answer) a logistic unit with the cross-entropy cost function
  - FWC - reason: b/c -t log (z) - (1 - t) log (1 - z) is equivalent to

  iii. The output of a neuro-probabilistic language model is a large softmax unit and this creates problems if the vocabulary size is large. Andy claims that the following method solves this problem: At every iteration of training, train the network to predict the current learned feature vector of the target word (instead of using a softmax). Since the embedding dimensionality is typically much smaller than the vocabulary size, we don't have the problem of having many output weights any more. Which of the following are correct? Check all that apply.
  - (check) If we add in extra derivatives that change the feature vector for the target word to be more like what is predicted, it may find a trivial solution in which all words have the same feature vector.
  - (check) The serialized version of the model discussed in the slides is using the current word embedding for the output word, but it's optimizing something different than what Andy is suggesting.
  - (not checked) In theory there's nothing wrong with Andy's idea. However, the number of learnable parameters will be so far reduced that the network no longer has sufficient learning capacity to do the task well.
  - (not checked) Andy is correct: this is equivalent to the serialized version of the model discussed in the lecture.

  iv. (a) optimal -> 4; (b) greedy -> 2

  v. No

  vi. In the Collobert and Weston model, the problem of learning a feature vector from a sequence of words is turned into a problem of: Learning a binary classifier.
  - FWC - (reason) kws: **dual** (see above for this word also)

  vii. not worried - network doesn't care about ordering
  - 7 (take 2) - network loses the location

## Week 5a: Why object recognition is difficult

- Segmentation
- Lighting
- Deformation
- 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.
  - FWC - this suggests videos of objects being used might be useful for image recognition
- Viewpoint/transformation
  - Imagine a medical database in which the age of the patient is sometimes labeled incorrectly as the patient's weight - this is called "dimension hopping" which needs to be eliminated before applying ML

## 5c: Convolutional neural networks for hand-written digit recognition

- The replicated feature approach (currently the dominant approach for NNs)
  - Use many different copies of the same feature detector w/ diff positions
    - Could also replicate across scale and orientation (tricky and expensive)
  - Use several different features tyeps, each w/ its own map of replicated features (FWC - each with its own convolution function)
- **Backpropagation with weight constraints**
  - It's easy to modify backprop to incorporate linear constraints btw weights
    - Start with $w_1 = w_2$, then at every iteration ensure that $\Delta w_1 = \Delta w_2$
  - Compute the gradients as usual, but then modify them so they satisfy constraints
    - set $\partial E/\partial w_1$ (and the same for $w_2$) to the average of the two partial derivatives
- Invariant knowledge: if a feature is useful in *some* locations during *training*, detectors for that feature will be available in all locations during *testing*.
  - "equivariance in the activities and invariance in the weights"
- Pooling
  - Achieve a small amount of translational invariance at each level by pooling (averaging or taking the max, which is slightly better) four neighboring replicated detectors to give a single output to the next level
  - Problem: after several levels of pooling, we've lost info about precise positioning of things
  - allows us to recognize if the image is a face "but if you want to recognize *whose* face it is" you need precise spatial relationships between high-level parts, which has been lost by CNNs
- 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**
  - Architecture of Le Net
    - C1 features maps - 6 at 28x28 pixels each, each pixel in one of these maps is computed by applying 3x3 convolution function to original image, but all 3x3 pools are the same, so there are only 9 parameters per map
    - S2 feature maps - "subsampling" == "pooling" to reduce each 28x28 down to 14x14
    - C3 feature maps 16 @ 10x10
    - S4 feature maps 16 @ 5x5
    - C5 layer (i.e. no more feature map, just a straight layer) of 120 nodes
    - F6 layer 84 (fully connected)
    - output 10 (one for each digit, Gaussian(/softmax?) fully connected)
- **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, e.g. build a simulator (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.
                - FWC - e.g. risk factors, residual variance, and phantom factors -- allows computation of model sensitivity to assumptions
            - **They can then train a large, deep, dumb net on a GPU without much overfitting**. -- only because they have so much extra training data
            - They achieve about 35 errors.
    - How to detect a significant drop in the error rate?
        - Is 30 out of 10,000 significantly better than 40?
        - Need to look at which errors first model got right but second got wrong and vice versa.
        - McNemar test: uses ratio of model_1_wrong_model_2_right to model_1_right_model_2_wrong
            - if 30 and 40 errors this could break down into [29 shared plus 1 vs. 11] or [15 shared plus 25 vs. 15]
- A neural network for ImageNet (16% vs. 26% for all other participants in the 2012 competition)
    - Alex Krizhevsky (NIPS 2012) developed a very deep CNN with the following architecture:
        - 7 hidden layers ("deeper than usual") not counting some max pooling layers.
        - The early layers were convolutional. "Could probably get away with using local receptive fields without tying any weights, but would need a much bigger computer. By making them convolutional, you cut down on the number of parameters a lot, and you cut down on the amount of training data a lot."
        - The last 2 layers were globally connected, which is where most parameters are ~16mm between those 2.
            - These 2 layers are looking for combinations of the features extracted by the earlier layers--and obviously there is combinatorially l
        - Activation functions:
            - **Rectified linear units** (ReLUs) in every hidden layer, which **train much faster and more expressive than logistic units (nobody uses logistic anymore)**
            - **Competitive normalization** to suppress hidden activities when nearby units have stronger activities, which helps w/ variations in intensity.
        - Train on random 224x224 patches of 256x256 images to get more data in addition to left-to-right reflections.
            - At test time, combine the opinions of 10 different 224x224 patches: 4 corners + center + 5 reflections
        - Use "**dropout**" to regularize the weights in the globally connected layers (worth several percentage points of improvement)
            - half of hidden units are randomly removed for each training example -> units cannot learn to overly correct for each other (FWC - can't be too co-linear)
            - **prevents overfitting**

## Lecture 6a: Overview of mini-batch gradient descent

- FWC - since the error surface lies in a space composed of pieces of quadratic bowls and the direction of steepest descent is only towards the minimum for perfect circle (cross sections), and for very skinny ellipses it is perpendicular, why not have a **normalization** procedure that attempts to make circles? (also see 'Shifting the inputs' slide in lecture 6b; also see 'separate adaptive learning rates' lecture 6d)
- SGD

- mini-batches (10, 100, 1000) are usually better than online b/c less computation updating weights
  - mini-batches need to be sampled in a way that they approx. the full distribution to prevent "sloshing" around in the quadratic bowl
- computing gradient for multiple cases simultaneously uses matrix mult which are very efficient on GPUs
- turn down the learning rate when the "error" stops decreasing
  - measure the "error" on a separate validation set

### Lecture 6b: Bag of tricks for mini-batch GD

- break symmetry by initializing w/ small random values
- shift inputs - demean each input component to prevent (101,101)->2 and (99,101)->0 (a skewed elliptical error surface)
  - htan = 2 * logistic - 1
- also helps to scale inputs to prevent (0.1,10)->1 and (0.1,-10)->0 (another skewed ellipse)
- more thorough method: decorrelation (guaranteed circles every time)
  - big win for linear neurons
  - e.g. PCA
- starting w/ big learning rate risks weights becoming all very large positive or negative while error derivatives become tiny
  - people usually think they've reached a local minimum, but this usually isn't true, you're just stuck out on the plateau
  - another plateau that looks like a local minumum is in classification nets to use the "best guessing strategy" - just guess 1 with P = the proportion of 1s that are seen in the data (FWC - this is like learning the intercept but nothing else so are there other local minima where say the intercept and one coefficient are learned?)

### Lecture 6c: The momentum (viscosity) model to GD

- if the error surface is a tilted plane the ball rolling down the plane will reach a terminal velocity when the incoming error gradient exactly balances the decay/viscosity/alpha of the previous gradients
- a big learning rate by itself towards the end of learning generates big divergent oscillations across the ravine (sloshing), momentum dampens this sloshing allowing for larger learning rates
- better momentum (Sutskever, 2012 inspired by Nesterov, 1983) - first make a big jump in direction of previously accumulated gradient, then measure gradient again where you end up and make a small correction (sliding scale EAFP)
  - "much better to gamble then make a correction, than to make a correction then gamble"

### Lecture 6c: The momentum (viscosity) model to GD

- add 0.05 if consecutive gradient signs the same, multiply by 0.95 if opposite sign
- can combine adaptive learning w/ 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 doesn't care about alignment of the axes, it can deal with diagonal ellipses (and going in diagonal direction quickly), which adaptive learning cannot do

### Lecture 6e: rmsprop: Divide the gradient by a running average of its recent magnitude

- rmsprop is an extension of rprop to tailor it for mini-batch learning (**Hinton's current favorite model**)
- 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, which makes it hard to choose a single global learning rate--and adaptive learning rates are tricky too.
    - This escapes from plateaus with tiny

    - rprop 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.
        - O/w decrease multiplicatively (e.g. 0.5)
        - limit step sizes to in [1e-6, 50]
    - rprop doesn't work well (lots of people have tried) for mini-batches b/c weights can grow too much in the presence of consecutive equal sized & signed gradients (e.g. 0.1 0.1 0.1 0.1 0.1 0.1 -0.9).
- rmsprop
    - rprop is equivalent to dividing the gradient by the magnitude of the gradient: g/|g| or g/sqrt(g^2)
        - the problem w/ mini-batch rprop is that we divide by a different # foreach mini-batch
        - so why not fix the # we divide by
    - rmsprop - keep a moving avg of the squared (RMS) gradient for each weight
        - MeanSquare(w,t) = 0.9 MeanSquare(w,t-1) + 0.1 [$\partial E/\partial w$ (t)]^2
        - dividing gradient by MeanSquare(w,t) makes learning work much better (Tijmen Tieleman, unpublished)
    - commentary on rmsprop combined w/ momentum (which doesn't seem to help as much)
- **Summary of learning methods for NNs**
    - For small datasets (e.g. 10k cases) or bigger *w/out* much redundancey, use *full-batch*
        - Conjugate gradient, LBFGS (packaged versions, simple for writing papers, no explanation of hyperparameter tweaking necessary)
        - adaptive learning rates, rprop
    - For bit, redundant datasets use *mini-batches*
        a. Try GD w/ momentum
        b. Try **rmsprop** (with momentum?)
        c. Try LeCun's latest recipe (e.g. "No more pesky learning rates" similar to rmsprop)
    - **Why is there no single answer?**
        - lots of different NNs (*esp. ones w/ narrow bottlenecks*)
        - recurrent, wide-shallow (can be optimized with not-very-accurate methods)
        - some require accurate weights, some don't
        - some have *many very rare cases* (e.g. words [FWC - stocks?])

### Quiz 6

1. WWWwww---
2. 2 checks

- too small learning rate
- large scale inputs (i.e. plateaus at logistic extremes)
- (unchecked--wrong!) large weight inits -> this will result in large corrective gradients right away (which is wrong, i.e. should've been checked)

3. circular cloud
4. the two monotonically descending L shaped red curves that don't get as low as blue (one that is steep at first, crosses the blue, then levels off (correct); the other that just never gets there (wrong)) -- should have selected the one that converges at first, then diverges
5. 2 check

- object detection (1e6 training cases, large dataset, but maybe not redundant?)
- speech recognition (large and redundant)
- (unchecked) sentiment analysis w/ 100 cases (small dataset)
- (unchecked) disease prediction (small dataset)

## 7a: Modeling sequences, a brief overview

- 2 standard models (not RNNs)

    i. Kalman Filtering (engineers love them!) - efficient recursive way of updating your representation of the hidden state [e.g. covariance matrix] given a new observation
        - given an output, we can't know for sure the hidden state, but we can estimate a Gaussian distribution over the possible hidden states
    ii. Hidden Markov Models (computer scientists love them) - have a discrete 1-of-N state, transitions btw states are stochastic and controlled by transition matrix, outputs are produced stochastically as well
        - this is where the term "hidden" layer comes from (coined by Hinton himself)
        - "there's an easy solution, based on dynamic programming, to take the observations we've made and from those, compute the probability distribution across the hidden states"
        - fundamental limitation of HMMs: with only only N states, they can only remember log(N) bits about what has been generated thus far [FWC - they aren't big enough; it's not possible to enumerate enough hidden states]
            - e.g. given 300 syntactic forms, 100k semantic types, and 1k combinations of voice type & intonation => 30e9 hidden states
- RNNs have a more efficient way of storing/representing information (they're also Turing-complete)
    i. distributed hidden states => efficiency
    ii. non-linear => updating hidden state in complicated ways
- RNNs are hard to train though because of their computational power

## 7b: Training RNNs with backpropagation

- "backpropagation through time algorithm"
- **feed-forward network with constrained weights*** - just "unroll" the network into a typical feed-forward net (with duplicated/constrained weights)
- remember: it's easy to modify backprop to incorporate linear constraints (compute gradients as usual, then modify them to satisfy the constraints)
- (1) forward pass to build up stack of activities at each time step followed by (2) backward pass to peel activities off the stack to compute error derivatives then (3) average derivatives from all different times to update weights
- Specifying the states of the same subset of the units at every time step is the natural way to model most sequential data.
- Specifying targets:
    i. specify desired final activities of all units
    ii. activities for the last few time steps (good for learning *attractors*, i.e. to have the net "settle down" towards the end; it's easy to average in errors as we backpropagate through the final layers)
    iii. specify desired activity of a subset of the outputs: natural way to train a network that should be producing "continuous output" (the other units are hidden or input)
- Q: Suppose we're training an RNN on a sequence of numbers. After it has seen all the numbers in the sequence, we want it to tell us the sum of all the numbers in the sequence. Which of the following statements are correct?
    - A1: To provide input, we should specify the state of one unit (say unit #1) at every time step. Reason: There's one input at each time step, the next number in the sequence.
    - A2: We should specify a target for one unit (say unit #2) only at the final time step. Reason: There's one output value, which occurs only at the last time step. That's where the model is expected to produce the sum of the numbers in the sequence.

## 7c: A toy example of training an RNN

- We could train a *feedforward* net to do binary addition, but there are obvious regularities that it cannot capture *efficiently*
    - would have to decide in advance how many digits
    - the processing applied to the beginning of a # wouldn't generalize to the end b/c it'd be using different weights
- "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)"**

## 7d: Why it is difficult to train an RNN

- exploding and vanishing gradients
- 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 [FWC - these squashing functions get applied at every single layer over and over]
  - The backward pass is completely linear (which most people find surprising). If you double the error derivatives at the final layer, all the error derivatives [at all layers throughout the net] will double.
    - backprop is a linear system which suffer from problem: when you iterate, the gradients explode or die
    - if small weights => they shrink exponentially [due to constraint averaging]; if big => they grow exponentially
- Typical feedforward NNs can cope with these exponential effects b/c they only have a few hidden layers.
  - In an RNN trained on long sequence (e.g. 100 time steps) the gradients can easily explode or vanish.
- 4 effective ways to learn an RNN
  i. LSTMs - compose RNN out of little modules designed to remember for long periods
  ii. Hessian Free Optimization - a fancy optimizer to detect directions w/ tiny gradient but even smaller curvature (The HF Optimizer, Martens and Sutskever, 2011, is good at this)
  iii. Echo State Networds - work around the problem via careful initialization, weakly coupled oscillators
  iv. Good init w/ momentum - ESN initialization, but then learn w/ momentum

## 7e: Long Short-Term Memory

- The dynamic state of a RNN is its short term memory, but we want to make the short term memory last for a long time.
- LSTMs have been successful for cursive handwriting recognition (Graves & Schmidhuber, 2009) - input: (x,y,p) coordinates for pen tip (where p is boolean = pen up/down) - output: seq of chars
- "keep"/forget, "write" (cell influences rest of net), and "read" (rest of net influences cell) gates - all logistics (**logistics are used because they have nice derivatives**)

### Quiz 7

1. 16 logistic hidden units can model 16 bits of information (WRONG, correct answer is ">16 bits")
2. RNN and FFNN, both w/ 200ms (WRONG, RNN w/ 30ms of input should also be selected)
3. see (green) notebook, A: -0.355 (CORRECT)
4. see (green) notebook, A: 0.01392 (CORRECT)
5. exploding (WRONG, correct "vanishing")
6. see (green) notebook, A: c (CORRECT)

- take 2 & 3
  i. 16 HMM units -> 16 bits (WRONG, correct answer is "4 bits" for an HMM)
  ii. (WRONG) asked this time for T=2 (T=1 was asked the last 2 times)

## 8A: A brief overview of "Hessian-Free" optimization (no video, but lecture slides are in lec8.pdf)

- **Good explanation of why (efficient) optimization involves multiplying by the inverse of a covariance matrix**
  - "maximum error reduction depends on the ratio of the gradient to the curvature, so a good direction to move in is one w/ high ratio of gradient to curvature, even if the gradient itself is small"

        - but if the error surface has circular cross-sections the gradient is fine
        - so apply a linear transformation to turn ellipses into circles
        - **Newton's method** multiplies the gradient vector by the **inverse of the curvature matrix (FWC - the covariance matrix?)**
            - Δw = -epsilon * H(w)^-1 * ∂E/∂w
        - on a real quadratic surface this jumps to the minimum in one step
        - there are too many terms in the curvature matrix H(w) to invert it though
        - in the HF method, approximate the curvature matrix, then minimize error using *conjugate gradient*
- **Conjugate gradient**
    - ensure each next direction is "conjugate" to the previous so that you don't oscillate/thrash too much
    - Also see 'non-linear conjugate gradient' for non-quadratic error surfaces (where it still works quite well)

## 8b: Modeling character strings with multiplicative connections

- The multiplicative factors described in the lecture are an alternative to simply letting the input character choose the hidden-to-hidden weight matrix.
- 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? They often have something to do w/ upper lip or nose: snot, snarl, snog. "Many people say, 'What about snow?' but ask yourself: why is 'snow' such a good word for cocaine?"
        - What about New York? One lexical item or 2? "New Yorkminster Roof"?
        - What about Finnish (and Agglutinative Language)? This word takes 5 words in English to say the same thing: ymmartamattomyydellansakaan (FWC lots of umlaut's left off this "word")
    - It's a lot easier to predict 86 chars than 100,000 words
    - 2 ways to build a NN
        a. 1500-hidden layer RNN; requires backprop to the beginning of the string
        b. a tree where each node is a hidden state vector (exponentially many nodes), but different nodes can share structure b/c they use distributed representations
        - e.g. if we arrive at a node "fix" the hidden state can encode that this is a verb and that 'i' or 'e' often follow ("fixed" or "fixing"), so can operate on , which can be shared by all the verbs -- and might follow the *conjunction* of followed by
- Multiplicative connections
    - Use the current char to choose the whole 1500x1500 hidden-to-hidden weight matrix, but constrain the matrices to be similar for each char by using **factors**!!!
    - 1000 hidden units & 86 character units => 2086 weights (1000 from previous hidden state, 86 from incoming char, plus 1000 outgoing for next hidden state)
        - vector of inputs to group c for factor f: c_f = (b'w_f)(a'u_f)v_f
            - b'w_f : scalar input to f from group b (e.g. 86 dim)
            - a'u_f : scalar input to f from group a (e.g. 1000 dim)
            - v_f : vector of output weights to be scaled by the 2 scalars above (1000 dim)
        - rearrange: c_f = (b'w_f)(u_f * v_f')(a)
            - u_f * v_f' : outer product transition matrix w/ rank 1
            - a : current hidden state gets multiplied to determine the input that factor f gives to next hidden state
    - a can be factored out : C = sum_f[(b'w_f)(u_f * v_f')] * a where the matrix sum multiplied by a is the transition matrix
    - see page 17 of lec8.pdf at file:///home/fred/Documents/articles/geoff_hinton's_machine_learning_coursera/lec8.pdf

## 8c: Learning to predict the next character using HF

- Ilya Sutskever used 5 million strings of 100 characters taken from wikipedia. For each string he starts predicting at the 11th character. (**FWC - attempt to recreate this**)
  - best single model for character prediction (combinations of many models do better)
  - start w/ model in default hidden state, give it a "burn-in" (FWC - ramp-in) sequence of chars and let it update its hiddens state after each char
  - See: 'How to generate character strings from the model' slide to see what it "knows" **(FWC - to generate ideas from it)**
    - tell it that whatever char it predicts is correct and let it go on generating **(FWC - this could be used to generate scenarios for monte carlo simulation. the analogy to character learning might be learning a mean and a stdev (and skew)--the moments of a distribution--rather than different buckets of means which wouldn't have any represented order to them)**
  - Also see: 'Some completions produced by the model' slide
    - "The meaning of life is *literary recognition.*" (6th try)
  - it learns loose semantics, but humans learn these things too
    - If you have to answer this question very quickly what do you answer "What do cows drink?"
- RNNs for predicting next word (as opposed to next char)
  - Tomas Mikolov (word2vec!) has recently trained quite large RNNs on large training sets using BPTT
  - better than feed-forward NNs, better than best other models, and even better when averaged w/ others
  - RNNs **require much less training data to reach the same level of performance**
    - FWC - this is because of their constraints (& factors), the other models have too many degrees of freedom?
  - RNNs also improve faster as the datasets get bigger

## 8d: Echo State Networks

- Big random (fixed) expansion of input vector, then learn the output layer with a linear model.
  - Similar to Support Vector Machines (SVMs) which just do this more efficiently
- Equivalent idea for RNNs is to randomize & fix the input->hidden connections and hidden->hidden connections and just learn the hidden->output
  - Will only work if you set the random connections very carefully so that the RNN doesn't explode or die
  - Set them so that the length (L2 norm) of the activity vector stays about the same length after each iteration aka so that the spectral radius is 1 (the biggest eigenvalue of the hidden->hidden matrix is 1 or it would be 1 if it were a linear system - we want to achieve the same property in a nonlinear system)
  - This allows the input to *echo* around the network for a long time
  - Also important to use sparse connectivity (lots of 0 weights and some big weights rather than lots of medium sized weights) - this creates lots of loosely coupled oscillators (information can hang around in one part of the net and not propagate to other parts too quickly)
  - Choose input->hidden scale very carefully which need to drive the loosely coupled oscillators w/out wiping out the information from the past that they already contain
  - Learning is very fast (fortunately) so we can experiment with the scales of these connections and level of sparseness (hyperparameter tuning)
- Example - predict a sine wave from its frequency
- **Impressive modeling of 1-dimensional time series** very far into the future
  - but aren't very good for high-dimensional data like pre-processed audio or video b/c they need many more hidden units than an RNN that learns its hidden->hidden weights
- Sukskever (2012) used ESN initialization in a normal RNN (with rmsprop and momentum) - very efficient/effective

### Week 8 Quiz

1. (checked) can use model where input char chooses whole matrix; (unchecked) can't use additive/obvious model (not sufficiently flexible); (unchecked) too many factors b/c simply choosing a multiplicative matrix for each char is at least as flexible as a factor-constrained matrix for each char; (checked WRONG - an additive model can't express a multiplicative one) can use additive with modification b/c the modification (one for each factor) acts like the multiplicative model
2. 1 - because still selecting a single matrix that connects each hidden->hidden (WRONG - 1 per factor, so the answer is 1000, b/c the hidden->hidden weights are only constrained by the factors in this scenario, not constrained to 1 like in the former scenario)
3. 3086000 (1500*2 + 86 for each of the 1000 factors) (see lecture notes above) (CORRECT)
4. It should learn eventually, but that requires more compute power than is available today (WRONG - that would be overfitting) Wrong: Basic calculations about the size of the hidden state vector show that the model can never learn to reliably generate any fixed string of text that's more than 38 chars long (38 is the sqrt of 1500, the number of hidden units) Correct: That would've been overfitting, which was carefully avoided.
5. No - they aren't at risk of overfitting b/c the hidden->hidden weights are fixed
6. Don't always use the single most likely char next. Do sample from the distribution. A probability distribution is better visualized by samples from it.
7. **In Echo State Networks, does it matter whether the hidden units are linear or logistic (or some other nonlinearity)? A: Yes. With linear hidden units, the output would become a linear function of the inputs, and we typically want to learn nonlinear functions of the input. Therefore, linear hidden units are a bad choice.**

## Lecture 9a: Overview of ways to improve generalization

- Network capacity can be controlled in several ways
  i. Architecture - limit # of hidden neurons
  ii. Early stopping - assumes real relationships are learned before spurious ones; start w/ very small weights (important!) and stop training when (you're sure) validation error performance starts getting worse (and then go back to when things were best)
  iii. Weight-decay - penalize large weights
  iv. Noise - add noise to weights or activities
- Cross-validation - a better way to choose meta parameters
  - 3 sets of data: training, validation, test (only used once)
- Why early stopping works
  - A network with small weights has lower capacity, but why? [FWC - this is kinda like degrees of freedom. Is there a way to combine the number of weights with their sizes to come up with a DF metric?]
  - When weights are small, if the hidden units are logistic units, they are in the linear range of the logistic function (slope = 0.25), so they behave very similarly to linear units.
  - So a small weight logistic network has no more capacity than a linear net.
  - [FWC - so then there is another point on the logistic curve where it is quadratic, cubic, etc. The average (and stdev) across all units of this metric might be a good measure of DF. Sigmoid/Logistic Power Series]
    - [FWC - we limit the weights when using other statistical approaches as well, e.g. SVD variants, but this seems to be a more intuitive reason for doing so.]

## Lecture 9b: Limiting the size of the weights

- Standard approach is to add a (constant times the) sum of the squared weights (L2) into the cost function--a penalty.
  - Large weights to occur only when there are large error derivatives (see slides)
  - Using L1 (absolute value) penalty is sometimes better b/c lots of weights are then 0, which makes a network easier to understand. Or even more extreme, sqrt(abs), makes escape from 0 difficult but then negligible effect on really large weights (allows for a few big weights).
- **Better idea: constrain the squared length of the weight vector**
  - If an update violates the constraints, then scale down the weight vector to the allowed length [FWC - this could be used for across-the-board constraints, rather than penalties,

on risk factor exposures]
    - This is much more effective at pushing irrelevant weights towards 0 b/c big weights, via the scaling, cause the small weights to get smaller. "The penalties are then just the LaGrange multipliers required to keep the constraints satisfied."
    - Hinton finds such **constraints to work "noticeably" better than penalties**
    - FWC - this is similar to mean-variance optimization with flat-bottomed parabolas
    - It's *much easier to set a sensible constraint than a sensible weight penalty*

## Lecture 9c: Using noise as a regularizer

- L2-weight penalty is equivalent to adding noise to the inputs (in a *linear* network)
    - Minimizing the squared error also minimizes the squared weights (because the noise *variance* gets scaled by the square of the weights), the second operand of this sum: y_j + N(0, w_i^2 * sigma_i^2)
    - E[(y - sum_i(w_i*e_i) - t)^2] = (y - t)^2 + sum_i(w_i^2 * sigma_i^2)
- Using noise in the activities as a regularizer
    - Make the units binary and stochastic on the forward pass, but then do the backward pass as if we had done the forward pass "properly"
        - In the forward pass choose 0 or 1 based on the "probabiliy" value of the logistic function.
    - This does worse on the training set (and trains considerably slower), but performance on the test set is significantly better (unpublished result)
        - [FWC - **so train with positive or negative returns, -1 or 1, but then compute cost (and propagate errors) with real-return-computed cost** -- and also note that not just tcosts, but liq constraints also, and all other *phantom* constraints will get baked into the learned function]
- [FWC - So adding noise to inputs, weights, and activations reduces overfitting (adds regularization), but what's the real mechanism for why? Does the added noise effectively dampen actual noise, making it more difficult to overfit? But then if you add too much noise, does it make real effects difficult to detect? Seems like you could slowly increase the amount of noise and measure validation performance along the way]

## Lecture 9d: Introduction to the Bayesian Approach

- "The main idea behind the Bayesian Approach is that instead of looking for the most likely setting of the parameters of a model, we should consider all possible settings of the parameters and try to figure out for each of those possible settings how probable [FWC - likely] it is, given the data we observed."
- POSTERIOR = PRIOR * DATA_LIKELIHOOD
    - With enough data, the likelihood term always "wins."
- If we flip a coin 100 times and see 53 heads then what is P(head)?
    - Frequentist answer (aka maximum likelihood) = 0.53
        - Set derivative d/dp of P(D) where P(D) = p^53 * (1-p)^47 to 0 (which is where p is *maximized*) => p = 0.53
        - Problems w/ MLE
            - If we only flip the coin once and get a head, then p = 1? No, 0.5 is still a better answer.
            - It's unreasonable to give a single answer. Instead lets say we're unsure about p.
    - See slide 24 (of 39) of lec9.pdf 'Using a distribution over parameter values' for how the posterior is updated after one coin flip given a uniform prior.
- Bayes
    - P(W|D) = P(W) * P(D|W) / P(D)
    - P(D) is a normalizing term (to ensure the distribution integral sums to 1) equal to integral_W[P(W) * P(D|W)] but for any P(W|D) in the equation above this is the same value--it doesn't depend on W b/c it's an integral over all possible Ws

## The Bayesian interpresation of weight decay/penalties

- Maximum a Posteriori Learning

    - Gives us a nice explanation of what's really going on when we use weight decay to control the capacity of a model
- Supervised Maximum Likelihood Learning
    - Finding the weight vector that minimizes the squared residuals is equivalent to finding a weight vector that *maximizes the log probability density of the correct answer*
    - see slide 30 of lec9.pdf for mathematical derivation
        - -log(P(t|y)) = k + (t-y)^2 / (2 * sigma^2)
            - note this is where the 2*sigma^2 in the denominator comes from
        - if -log(P(t|y)) is our cost function, this turns into minimizing a squared distance (the RHS)
        - *Minimizing squared error is the same as maximizing log probability under Gaussian distribution!* ("helpful to know" b/c when you're minimizing sq error you can make a probabilistic interpretation of what's going on)
- 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 P(D) doesn't depend on W, so doesn't affect the minimization
        - log P(D|W) is the log prob of target values given weights, W (normalized by 2*sigma_D^2 of the errors, the data)
        - log P(W) is the log prob of W under the prior (normalized by 2*sigma_W^2 of the weights)
    - Minimizing the squared weights is equivalent to maximizing the log probability of the weights under a *zero-mean* Gaussian prior (the same as for minimizing the squared error!)
    - multiply through by 2*sigma_D^2 gives us:
        - Cost = MSE + sigma_D^2 / sigma_W^2 * sum_i[w_i]
        - so we have a specific number for the weight penalty, the ratio of 2 variances (FWC - like a beta!), it's not an arbitrary choice as in previous lecture

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

- Estimating the variance of the Gaussian prior on the weights
    - After learning a model with some initial choice of variance for the weight prior, sigma_W^2, we could do a dirty trick called "empirical Bayes"
    - Set the variance of the Gaussian prior, sigma_W^2, 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 really violates a lot of the presuppositions of the Bayesian approach. We're using our *data* to decide what our beliefs are."
        - Fit a zero-mean Gaussian to the 1-dimensional distribution of the learned weight vals
        - => **we could easily learn different variances for diff sets of wgts (a benefit!)**
    - We don't need a validation set! which, to use it, would be very time consuming
- MacKay's quick and dirty method of choosing the ratio of the noise variance, sigma_D^2, to the weight prior variance, sigma_W^2
    - Start with guesses for both the noise variance and the weight prior variance (really just guess their ratio)
    - Loop/repeat:
        a. Do some learning using the ratio of the variances as the weight penalty coefficient
        b. Reset the noise variance to be the variance of the residual errors
        c. Reset the weight prior variance to be the variance of the distribution of the actual learned weights
    - **This works quite well in practice and MacKay won several competitions this way.**
- Q: Suppose we're using MacKay's method for setting weight costs. For one particular input unit, we decide to use the same prior variance for all of its outgoing weights to the hidden layer. Now suppose that the state of that input unit does not contain any useful information for getting the output right. Assuming we start with weights drawn from the prior distribution, which of the following statements is true?

    - FALSE (b/c weights are known to be 0 so variance shrinks) - After training the weights for a while without updating the weight prior variance, we expect the actual variance of the outgoing weights of that unit to be bigger than the prior variance.
    - TRUE - Every time we update the prior variance after doing some more learning, it will get smaller.
        - correct - Since that input unit contains no useful information, we expect the *error derivative to be small. The Gaussian weight prior will always push the weights towards 0 unless it is opposed by the error derivative.*

### Week 9 Quiz

1. the one where the training error reaches 0 is an overfit (not checked)
2. INCORRECT - "adding weight noise" is equivalent to "L2 regularization" so it's not either of those. it must be "L1 regularization" b/c something was used, though it should really be that a constraint on the size of the weight vector was used because a few large values happened
3. selected the "mustache" shape in order to have lot of weights close to 0 but a few a long way from 0
4. E = SSE, C = student-t cost = lambda/2 * sum_i[log(1 + w_i^2_], E_tot = E + C, what is d/dw E_tot

- A: lambda * w_i in the numerator of the C derivative

5. Different regularization methods have different effects on the learning process. For example (a) L2 regularization penalizes high weight values. (b) L1 regularization penalizes weight values that do not equal zero. Adding noise (c) to the weights during learning ensures that the learned hidden representations take extreme values. Sampling (d) the hidden representations regularizes the network by pushing the hidden representation to be binary during the forward pass which limits the modeling capacity of the network.

- Q: Given the shown [bimodal at -10 and 10] histogram of activations (just before the nonlinear logistic nonlinearity) for a Neural Network, what is the regularization method that has been used (check all that apply)?
- checked - 'adding noise to the weights' and 'sampling the hidden repr'
- unchecked - L1 and L2

6. INCORRECT - better on both training and test (the correct answer is worse on training and better on test)

## Week 9 programming assignment #3

- "The program checks your gradient computation for you, using a finite difference approximation to the gradient. If that finite difference approximation results in an approximate gradient that's very different from what your gradient computation procedure produced, then the program prints an error message. This is hugely helpful debugging information. Imagine that you have the gradient computation done wrong, but you don't have such a sanity check: your optimization would probably fail in many weird and wonderful ways, and you'd be worrying that perhaps you picked a bad learning rate or so. (In fact, that's exactly what happened to me when I was preparing this assignment, before I had the **gradient checker** working.) With a finite difference gradient checker, at least you'll know that you probably got the gradient right. It's all approximate, so the checker can never know for sure that you did it right, but if your gradient computation is seriously wrong, the checker will probably notice."

## Lecture 10a: Why it helps to combine models

- Also see XGBoost notes on Machine Learning wiki page
- "If we have a single model we have to choose some capacity for it. If we choose too little capacity, it won't be able to fit the regularities in the training data. If we choose too much capacity, it will be able to fit the sampling error in the training set data. By averaging many models we can get a better tradeoff between fitting too few regularities and overfitting the sampling error in the data. This effect is largest when the models make very different predictions from each other."
- Combining networks: the bias-variance tradeoff

    - When the amount of training data is limited, we get overfitting.
        - Averaging the predictions of many different (kinds of) models (e.g. SVM, AdaBoost, NN) 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.
            - It's called "variance" because if we were to get another training set of the same size from our distribution, our model would fit differently to that training set because it has different sampling error, so we'll get **variance in the way the models fit to different training sets [FWC - e.g. variance in R^2].**
        - By averaging away the variance we can use individual models with high capacity. These models have high variance but low bias.
        - We can get low bias without getting high **variance [FWC - overfitting]** by using averaging to get rid of the high variance.
- 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) [FWC - make individual predictors uncorrelated with each other]
- Combining two networks reduces variance
    - The expected squared error we get, by picking a [single] model at random, is greater than the squared error we get by averaging the models by [exactly] the **variance of the outputs of the models**
    - E_i[(t-y_i)^2] = (t-E[y])^2 + **E[(y-E[y])^2]** - 2(t-E[y])(y-E[y]) [<- this last term vanishes because we expect the errors to be uncorrelated]
    - This only works if the noise is Gaussian
        - Don't try averaging a bunch of clocks because some may be approximately correct, but some may have stopped and be wildly wrong [skew and kurtosis matter]
        - Same applies to discrete distributions over class labels
            - if we have 2 models, i and j, that give the correct label probabilities, p_i and p_j
            - log((p_i+p_j)/2) >= (log(p_i)+log(p_j))/2 [think of the plot of the log function]
- Lots of ways to make predictors/models differ
- Making models differ by changing their training data
    - Bagging - Train diff models on diff subsets of the data (with replacement) ["bootstrapping"?]
    - Random forests - Use lots of different decision trees trained using bagging. They work well.
    - We could use bagging w/ NNs but its *very* expensive to train
    - Boosting - Train a seq of low capacity models. Weight the training cases differently for each model in the seq. Up-weight cases the previous models got wrong. [FWC - prone to overfitting]

## Lecture 10b: Mixture of Experts

- **Multi-regime**, train a net on each regime
- In boosting, the mixture of models depends on particular input cases.
- Can we do better that 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* [FWC - **regimes**]
    - 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. [FWC - this is similar to k-means clustering where the clusters separate from each other over time]
- Spectrum of models
    - Very local models (e.g. nearest neighbors)
        - very fast to fit (e.g. just store training cases)
    - Fully global models (e.g. polynomial)
        - may be slow to fit and also unstable (each param depends on all the data, so 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 we might not know in advance what defines 'different states of the economy'--we'll have to learn that too."**
        - So how do we partition the data into regimes?
- **Partitioning based on input alone versus partitioning based on the input-output relationship** [FWC - I tried this sort of clustering once at HBK in an Excel spreadsheet wrt SRs but didn't get anywhere]
    - 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. [FWC - think scatterplot smoothing (one model per datapoint) or computing a different model for different buckets of the data]
- An error function that encourages *cooperation*
    - Loss = (t-E[y_i])^2
    - This will overfit badly because models will learn to "fix up" errors that other models make.
    - Averaging models *during training* causes cooperation, not specialization
    - If there's a model that disagrees with the average (e.g. underestimates the target while the average overestimates) then do we really want to move the disagreeing model away from the target in order to "compensate" for all of the other models (to make the average model closer to the target)? Intuitively it seems better to move the disagreeing model towards the target.
- An error function that encourages *specialization*
    - If we want to encourage specialization we compare each predictor/model separately with the target.
    - We also use a "manager" to determine the probability of picking each expert (aka weight each model)
    - Loss = E[ p_i (t-E[y_i)^2 ]
    - Most experts/models/predictors end up ignoring most targets (due to fitted p_i)
- The mixture of experts architecture (*almost*)
    - The obvious/intuitive architecture: p_i and y_i as outputs
        - The p_i can be chosen with their very own sub-network, the manager, a *gating* network
    - p_i = exp(x_i) / sum_j[exp(x_j)]
    - Loss = E[ p_i (t-E[y_i)^2 ]
    - del Loss / del y_i = p_i (t-E[y_i)
    - If the manager, p_i, decides that there's a small probability of picking that expert for that training case, we get a very small gradient (and parameters inside that expert won't get disturbed)

    - *We want to increase p_i for all experts that give below average (weighted by p_i) squared error across all i* [see del Loss / del x_i below]
    - If we differentiate w.r.t. the outputs of the gating network (wrt the input to the softmax, which is called the "logit") we get a signal for training the gating net.
        - del Loss / del x_i = p_i ((t-y_i)^2 - Loss)
            - This gradient will increase p_i for all experts that produce below avg SE
            - This is what causes specialization.
    - There's a better cost function though.
- **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} or a "mixing rate"
    - Maximize the log probability of the target value under this mixture [FWC - sum] of Gaussians model (i.e. the sum of the two scaled Gaussians).
    - <P of target val on case c given Mixture of Experts> =
        - sum_i[ <mixing proportion i for case c, p_ic> exp(-0.5(t_c-y_ic)^2) / <normalization term for Gaussian w/ sig^2=1, sqrt(2pi)> ]
    - P(t_c | MoE) = sum_i[ p_ic exp(-0.5(t_c-y_ic)^2) / sqrt(2pi) ]

## Lecture 10c: The idea of full Bayesian learning

- 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, as shown in a previous lecture).
    - 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?
    - **A frequentist would say: If you do not have much data, you should use a simple model, because a complex one will overfit.** [FWC - financial people are all frequentists]
        - 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.
    - Bayesian learning means learning the posterior distribution: **P(omega | data)**. The kind of learning we saw earlier in this course is called Maximum Liklihood (ML) learning where we learn a set of parameters, omega, that maximizes P(data | omega)
- A classic example of overfitting
    - Which model do you believe, a line or a 5th order polynomial? Clearly the line b/c the complicated model, even though it fits the training data better, 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.
    - From a Bayesian perspective, **there is no reason why the amount of data should influence our prior beliefs about the complexity of the model.** [FWC - there is no reason why a low signal-to-noise ratio should influence our prior beliefs...etc...]
- 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. We're not following a path, we're only evaluating points.
    - 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 Maximum Liklihood learning (or MAP) when the posterior is vague or multimodal (this happens when data is scarce).
    - [FWC - This seems to be a dual/transpose of a grid over the input space with a distribution for each input. Both approaches yield a (or can be used to estimate the) distribution of outputs. Could the two approaches be combined? A simultaneous co-grid over both input and parameter space. But how would one use that?]
    - p(t_test | input_test) = sum_{g in grid}[ p(W_g | D) p(t_test | input_test, W_g) ]
- An example of full Bayesian learning
    - A neural net with 2 inputs, 1 output and 6 parameters
    - Allow each of the 6 weights or biases to have the 9 possible values => 9^6 grid-points (lots but not impossible [FWC - but yeah, lots!])
    - For each grid point, compute the P of the observed outputs over all *training cases*
    - Multiply prior for each grid point by the likelihood term and renormalize to get the posterior P for each
    - Make predictions (on test data) by using the *posterior Ps* to average the predictions made by the different grid points

## Lecture 10d: Making full Bayesian learning practical

- Markov Chain Monte Carlo biased in the direction of the gradient
- Sample weight vectors in proportion to their probability in the posterior distribution
- What can we do if there are too many parameters for a grid?
    - Only a tiny fraction of the grid points make a significant contribution to the predictions--lots of 0 posterior Ps.
    - An idea that makes Bayesian learning feasible: It might be good enough to just sample weight vectors according to their posterior probabilities.
    - <P output_test given D> = sum_{i in weight space}[ <posterior P_i> <probability distribution of output_test given W_i>]
    - p(y_test | input_test, D) = sum_i[ p(W_i | D) p(y_test | input_test, W_i) ]
    - Instead of adding up all terms in this sum, sample weight vectors according to p(W_i | D)
- Sampling weight vectors
    - In standard backpropagation we keep moving the weights in the direction that decreases the cost.
    - Eventually, the weights settle into a local minimum (or get stuck on plateau, or just move so slowly 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**" (MCMC) method. [FWC - "markov chain" because it has discrete states but "monte carlo" because it only has a sampling of them]
    - **MCMC makes it feasible to use full Bayesian learning with thousands of parameters.**
    - L'orangian or Langevin (sp?) Method, not the most efficient

    - 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!** (very clever)
    - Ahn, Korattikara & Welling (ICML 2012) showed how to do this fairly efficiently.
        - file:///home/fred/Documents/articles/bayesian_posterior_sampling_via_mini_batch_ahn_krattikara_welling_2012_782.pdf
        - "an algorithm as simple as stochastic gradient descent is almost optimally efficient. We therefore argue that for Bayesian methods to remain useful in an age when the datasets grow at an exponential rate, they need to embrace the ideas of the stochastic optimization literature."
        - "Our main claim is therefore that we can trade-in a usually small bias in our estimate of the posterior distribution against a potentially very large computational gain, which could in turn be used to draw more samples and reduce sampling variance."
        - "it is an efficient optimization algorithm that smoothly turns into a sampler when the correct (statistical) scale of precision is reached."
    - So full Bayesian learning is now possible with lots of parameters.

## Lecture 10e: Dropout: an efficient way to combine neural nets

- Method of combining a very large number of NN models w/out having to train them all.
- Two ways to average models: (1) mixture (arithmetic mean) and (2) product (geometric mean renormalized to sum to 1)
- **Dropout**: An efficient way to average many large neural nets (http://arxiv.org/abs/1207.0580)
    - **An alternative to doing the correct Bayesian thing**. Probably doens't work quite as well, but much more practical [FWC - assuming we're starting with NNs to begin with]
    - 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 (provided we're using a softmax output group).
- 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 (in a multi-layer net), but it's a pretty good approximation, and it's fast.
    - Alternatively, run the (lots of) stochastic model several times on the same input (with dropout and then average across those stochastic models). [FWC - does he mean this is similar to "sampling many different architectures and taking the geometric mean" as mentioned above?]
        - **Benefit: 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 "and it works very well."
        - Presentation: Denoising Autoencoders (file:///home/fred/Documents/articles/overfitting/denoising_autoencoder_presentation_pascal_vincent_part_2.pdf)
        - Think of classical autoencoder in overcomplete case: d'>=d
        - *Perfect reconstruction is possible without having learnt anything useful!*
        - Denoising autoencoder learns useful representation in this case.
        - Being good at denoising requires capturing structure in the input.
        - **Denoising autoencoder can be seen as a way to learn a manifold** (p. 31) [FWC - scatterplot smoother]
        - *Intermediate/hidden representation Y can be interpreted as a coordinate system for points on (FWC - or w.r.t.?) the manifold* [FWC - a better solution to the "banana problem"]
        - It can be shown that *minimizing the expected reconstruction error* amounts to *maximizing a lower bound on mutual information*
        - Denoising autoencoder training can thus be justified by the objective that hidden representation Y captures as much information as possible about X even as Y is a function of corrupted input.
        - Unsupervised initialization of layers with an explicit denoising criterion appears to help capture interesting structure in the input distribution.
        - This leads to **intermediate representations much better suited for subsequent learning tasks such as supervised classification**.
        - Resulting algorithm for learning deep networks is simple and improves on state-of-the-art on benchmark problems.
        - We are currently investigating the effect of different types of corruption process, and applying the technique to recurrent nets.
- 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
    - FWC idea - pairwise dropout, let certain pairs of neurons always exist or be dropped out together to allow *some* complex relationships
    - 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 (with lots of players) are not robust (someone's going to tell--better to have lots of little conspiracies)
    - **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.*

## Lecture 10 Quiz

1. When learning a mixture of experts, it is desirable that each expert specializes in a different area of the input space. But then at test time, how will we know which expert to use?

- CHECKED - We also learn a "manager" model that sees the input and assigns probabilities for picking each expert. We then get predictions from all the experts and take their weighted average using the probabilities.

    - Correct - A Mixture of Experts can be seen as input-dependent model averaging. The input is used to decide how much weight should be assigned to each model and then a weighted average is taken.
- We also learn a "manager" model that sees the input and assigns probabilities for picking each expert. We then choose the expert that has the highest probability and use it to make a prediction.
- We see which training case the test case is closest to (in input space) and use the model that was used for that training case.
- We uniformly average the predictions of each expert.

2. Which data set can not be classified perfectly with a linear classifier but can be classified perfectly with a mixture of two experts where each expert is a linear classifier ? Assume that the manager is also linear, i.e., it can decide which expert to use based on a linear criterion only. (In other words, the manager has a linear function f and given any input case x, it must decide to apply expert 1 with probability 1 if $f(x)>0$ and expert 2 with probability 1 if $f(x)\leq 0$.
3. Andy has a dataset of points that he wishes to classify. This set is shown below. Being knowledgable about bagging, he samples this data set and creates 3 separate ones. He then uses a neural net to learn separate classifiers on each data set. The learned classifier boundaries are shown below. Note that each data set is a subset of the complete dataset. Which of the following statements is true?

- All the learned models make a lot of errors ("high-bias"), so model averaging is unlikely to help in generalization.
- All the learned models make a lot of errors ("high-bias"), so model averaging is likely to help in generalization.
- The learned models are different ("high-variance") and do well on their training sets, so model averaging is unlikely to help in generalization.
- CHECKED - The learned models are different ("high-variance") and do well on their training sets, so model averaging is likely to help in generalization. Correct - The classifiers are fairly different, but overfitted to their training sets. Averaging them could lead to a more generalizable model.

4. In Bayesian learning, we learn a probability distribution over parameters of the model. Then at test time, how should this distribution be used to get predictions with the highest possible accuracy?

- Sample the distribution once to get a parameter setting and use it to make a prediction.
- Pick the parameter setting that has maximum probability and use it to make a prediction.
- Sample a lot of parameters using some sampling procedure (such as MCMC) and average the parameters. Then use the averaged parameter setting to obtain a prediction.
- CHECKED - Sample a lot of parameters using some sampling procedure (such as MCMC) and average the predictions obtained by using each parameter setting separately.
    - Correct - This method makes sure that we use a lot of models and also choose the models in proportion to how much we can trust them.

5. Amy is trying different MCMC samplers to sample from a probability distribution. Each option shows a few samples obtained by running a sampler. It is known that the distribution is multimodal and peaked. Which among the following is the best sampler ? Correct - This sampler can access multiple modes and does not give a lot of samples from low probability regions. This is what a good sampler is expected to do.
6. Brian wants to learn a classifier to predict if a movie is "good" or "bad" given its review. He has a lot of computational power and wants to use a very powerful model. However, he only has a small dataset of labelled movie reviews. He tried learning a massive neural net and found that it achieves zero training error very easily, but its test performance is much worse. He then trained a small neural net and found that it does not get zero training error, but still the test performance is no better than what the big model got. Neither a big nor a small model works for him! He is completely disappointed with neural nets now. He is willing to spend as much computational power as needed during training and testing. What suggestion can you give to help him?

- CHECKED - Train the big neural net with dropout in the hidden units.
    - Correct - Adding drop out helps prevent overfitting in large nets.

- CHECKED - Train many different models - neural nets, SVMs, decision trees - and average their predictions.
  - Correct - Model averaging with different kinds of models is useful because each model is likely to be different and make different errors.
- Train lots of small neural nets of the same architecture on the whole data and average their predictions.
- Look for a better optimization algorithm to help the large neural net.

## Lecture 11a: Hopfield Nets

- Sometimes called "energy based models" b/c their properties derive from a global energy function
  - One of the main reasons for the resurgence in machine learning
- A Hopfield net is composed of binary threshold units with recurrent connections between them.
  - if the connections are *symmetric* (FWC - i.e. not directed, as in a DAG), there is a global 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*:
    - "energy is bad, hence the upcoming negative signs"
    - **E = -sum_i(s_i*b_i) - sum_{i<j}(s_i*s_j*w_ij)**
    - The s'es are all binary, 0 or 1 (so the state can be thought of as the corners of a hypercube). The w's are not.
    - **FWC - this is the formula for energy (or negative "goodness") of an RBM also except that in an RBM, with its bipartite graph, you cross product the hidden_state with the visible state and then dot product that with the weights matrix** (and then divide by the number of configurations/particles; see configuration_goodness.m in programming assignment 4)
  - This simple quadratic energy function makes it possible for each unit to compute *locally* how it's state affects the global energy
    - Energy gap "is the difference in the global configuration depending whether or not i is on"
    - Energy gap = ΔE_i = E(s_i=0) - E(s_i=1) = b_i + sum_j(s_j*w_ij)
    - It's just a derivative (which, yes, seems like a silly thing to do b/c s'es are binary): del E / del s_i = b_i + sum_j(s_j*w_ij) -- without the negative signs b/c they're for going downhill
- Settling to an energy minimum
  - For each unit, chosen randomly, one at a time, figure out which of its two states gives a lower global energy, and put it in that state. This is equivalent to saying "use the binary threshold decision rule."
  - "easier to think about negative energies, -E, which I'll call 'goodness'"
- Example with a net containing 2 triangles that "hate" each other, both are minima, but one is local and one global.
- Why do the decisions need to be sequential?
  - If units make *simultaneous* decisions the energy could go up.
  - However, if the updates occur in parallel but with random timing, the oscillations are usually destroyed.
- 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.
  - 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 (Before Google)... i.e. because "search" is knowing only part of the content
  - Robust against hardware damage
  - Like reconstructing a dinosaur from a few fossils (psychology analogy)
    - Solving for variables in a series of equations (FWC)
- Storing memories in a Hopfield net

    - If we use activities of 1 and -1, as opposed to 0 and 1, we can *store a binary state vector* by incrementing the weight between any two units by the product of their activities:
        - Δw_ij = s_i * s_j
    - Slightly more complicated w/ 0 and 1:
        - Δw_ij = 4(s_i-0.5)(s_j-0.5)
        - FWC - Does the 4 come from the fact that (1,1) occurs 1/4 of the time: (1,1), (0,1), (1,0), (0,0)?
    - Very simple rule that is not error-driven: both its strength (can be computed in true online fashion) and its weakness (not very efficient storage [FWC - lots of redundant info])
    - We treat biases as weights from a permanently on unit.

## Lecture 11b: Dealing with spurious minima in Hopfield Nets

- Storage capacity of Hopfield nets limited by spurious minima
- 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, which does not make efficient use of the bits required to store the weights.
    - After storing M memories, each connection *weight* has an integer value in the range [-M, M] (*because we increase it by 1 or decrease it by 1 each time we store a memory, assuming we use states of -1/1), so the number of bits required to store the weights and biases is: N^2 log(2M+1) -- assuming all weights are equi-probable (i.e. no compression)
- 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 merging is what limits the capacity.
- 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 (the opposite of the storage rule) [**FWC - this is like my "container-not" learning theory where you start out with a new concept being represented by everything it's not**]
        - This will get rid of deep, spurious minima and increase memory capacity
        - They showed this worked, but they had no analysis (couldn't explain)
    - Crick and Mitchison proposed **unlearning** as a model of what dreams (REM sleep) are for.
        - That's why you don't remember them (unless you wake up during the dream)
        - So we're simply not storing what we're dreaming about, but why?
    - How much unlearning should we do? (a more mathematical problem)
        - 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. I.e. postdoc physicists can get a job in neuroscience if they can't find one in physics.
    - Instead of trying to store vectors in one shot, cycle through training set multiple times
        - Use the perceptron convergence procedure to train each unit given the states of all the other units in that vector
        - Statisticians call this technique "pseudo-likelihood" -- "get one thing right, given all the other things" -- main difference: in Hopfield Net, weights are symmetric, so we need to get 2 sets of weights and avg them

## Lecture 11c: Hopfield Nets with hidden units

- Weights on connections represent constraints on good interpretations, and by finding a low energy state, we find a good interpretation of the input vector.
- A different computational role for Hopfield nets
    - see picture on slide 17 of file:///home/fred/Documents/articles/geoff_hinton's_machine_learning_coursera/lec11.pdf
    - Instead of using the net to store memories, use it to construct interpretations [FWC - representations?] 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 [FWC - interpret] about 3D edges from 2D lines in an image?
    - A 2D line in an image could have been caused by many different (a family of) 3-D edges in the world.
    - There are 2 degrees of freedom missing in the image that exist in the world, namely the depth of the 2 endpoints of the line.
    - How can we use the low energy states of binary units to find interpretations of sensory input?
- Analogy to physics
    - There are only O(n^2) possible configurations of a volume of size O(n^3). Entropy is proportional to the horizon area.
    - This is because there are linkages/relationships between the various 3D configurations that must be satisfied (but what are they?)
    - If we have one "2D line" unit for each line in a picture, and one "3D line" unit for each possible interpretation of each "2D line" unit, then we can make the "3D line" interpretations *support* each other if their endpoints match in the picture (and *strongly support* if they join at right angles because we have knowledge about our world).
    - A Necker Cube thus has 2 low energy states, one for each front/back of the cube being in either the front or the back.
- Two difficult computational 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?
    - Notice we don't have a supervisor anywhere, we're just showing it inputs and asking it to construct sensible interpretations.

## Lecture 11d: Using stochastic units to improve search

- Explain how adding noise can let systems escape from local minima
- Noisy networks find better energy minima
    - A Hopfield net always makes decisions that reduce the energy, which 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)
    - Effectively, temperature flattens the energy landscape making it easier to escape local minima
    - Example:
        - A := local minima, B := global minima
        - High temperature transition probabilities: p(A->B)=0.2, p(B->A)=0.1
        - Low temperature: p(A->B)=0.001, p(B->A)=0.000001
            - probability gets smaller, but ratio gets better (2 to 1000)
        - But if we run for a long time at low temperature, convergence will take forever.
            - Good compromise: start high and gradually reduce (simulated annealing)
- 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.
    - $p(s_i=1) = 1 / (1 + \exp(-\Delta E/T))$
        - so when temperature, T, is high, these probabilities are all about 1/2
        - as we lower the temperature, depending on the sign of $\Delta E$, the unit will become more and more firmly on or off

        - at T=0, which is what we use in a Hopfield net, the sign of ΔE determines if the RHS goes to 0 or 1
- Simulated annealing is a distraction
    - 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 not be covered in this course).
    - From now on, we will use binary stochastic units that have a temperature of T=1
- Thermal equilibrium at a temperature of 1
    - Difficult concept!
        - Does NOT mean that the system has settled down into the lowest energy config
        - The thing that settles down is the *probability distribution* over configurations
        - Which settles to the *stationary distribution*
    - Nice intuitive way to think about it
        - Imagine a huge ensemble of systems that all have exactly the same energy fn (a very large number of stochastic Hopfield nets, all with the same weights)
        - The probability of a configuration is just the fraction of the systems that have that config
        - FWC - this is, again, like the difference between a grid over inputs vs. a (Bayesian) grid over parameters (configs)
        - FWC - like in Sean Carroll's book: at a given energy in a box how many configs of the particles in that box are there?
- 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 (PDF: all 0s except for one 1)
        - Or with an equal number of systems in each possible configuration (PDF: uniform)
    - 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 (because we have many, many more systems than configs)
- 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 (A, K, Q, etc.) 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.
        - In general we're interested in understanding systems where some configurations have lower energy than others.

## Lecture 11e: How a Boltzmann Machine models data

- It models binary data vectors [**FWC - models truths, true/falses**]
- Boltzmann Machines == Stochastic Hopfield nets with hidden units
- Modeling binary data
    - Given a training set of binary vectors, fit a model that will assign a probability to every possible binary vector.
    - 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
        - E.g. if you have a nuclear power plant and you want to detect if it's in an abnormal state (and thus a chance of blowing up) you want to be able to detect that w/out supervised learning (b/c you don't have any examples of blowing up)
        - **FWC - wouldn't really be necessary for anomaly detection of company characteristics b/c we have supervised learning for that (we know when there's an anomaly based on trading volume/p(info))... but perhaps that's actually not true; perhaps this could be a better early warning signal before information is known to the market**
    - 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(model_i | data) = p(data | model_i) / sum_j[p(data | model_j)]
- How a causal model generates data
    - First generate states of some latent (hidden) variables and then use those to generate the binary data
    - In a causal-generative model we generate data in two sequential steps
        - First pick the latent/hidden, h, 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) ]
    - "**Factor analysis, for example, is a causal model that uses continuous variables**"
- How a Boltzmann Machine generates data
    - A different generative model; *not* a causal-generative model
        - FWC - See slide 40 of lec12.pdf. This must be how they generate Picasso-looking art from non-Picasso art. Analogous to training on '2's and then applying to '3's.
    - 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 2 ways (that both agree w/ each other).
        a. Simply define the probability as follows: p(v,h) exp(-E(v,h))
        b. Or 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
- The (negative) Energy of a joint configuration
    - -E(v,h) = <visible-biases> + <hidden-biases> + <vis-vis-weights> + <vis-hid-weights> + <hid-hid-weights>
    - -E(v,h) = sum_i[v_i*b_i]* + *sum_k[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]
- Using energies to define probabilities
    - from programming assignment 4: "formula for the Boltzmann distribution (the probabality [sic] of a particular configuration of an RBM)"
    - 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.
        - p(v,h) = exp(-E(v,h)) / sum_{u,g}[exp(-E(u,g))] ... physicists call the denominator the "partition function" (note that it has exponential number of terms)
    - The probability of a configuration of the *visible units* [only] is the sum of the probabilities of all the joint configurations that contain it.
        - p(v) = sum_h[exp(-E(v,h))) / sum_{u,g}[exp(-E(u,g))]
- An example of how weights define a distribution
    - example shows how to, given all possible v-h pairs -> compute -E -> compute exp(-E) -> p(v,h) -> p(v)
    - slide 35 of lec11.pdf
- Getting a (global, v & h) 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 (MCMC) to get samples from the model starting from a random global configuration and keep picking units at random allowing them to stochastically update their states based on 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
        - $p(v,h) \propto \exp(-E(v,h))$
- Getting a sample from the posterior distribution over hidden configurations *for a given (visible) data vector* (which is needed for learning)
    - 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* [FWC - b/c they are given] 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
    - The reason we want to get samples from the posterior distribution, given a data vector, is we might want to know a good explanation for the observed data (anomaly?) and we might want to base our actions on that good explanation [FWC - p(info)?] but we also need to know this for learning.

## Lecture 11 Quiz

1. If $\Delta E=3$, then:

- P(s=1) increases when T increases.
- P(s=1) decreases when T increases. - CHECKED (b/c increasing T will increase $\Delta E$ which is P(s=0)-P(s=1) so decreasing in P(s=1))

2. The Hopfield network shown below has two visible units: V1 and V2. It has a connection between the two units, and each unit has a bias. Let W12=-10, b1=1, and b2=1 and the initial states of V1=0 and V2=0. If the network always updates both units simultaneously, then what is the lowest energy value that it will encounter (given those initial states)? If the network always updates the units one at a time, i.e. it alternates between updating V1 and updating V2, then what is the lowest energy value that it will encounter (given those initial states)? Write those two numbers with a comma between them. For example, if you think that the answer to that first question is 4, and that the answer to the second question is -7, then write this: 4, -7

- A: 0,-1 (INCORRECT, needs a space after the comma

3. This question is about Boltzmann Machines, a.k.a. a stochastic Hopfield networks. Recall from the lecture that when we pick a new state si for unit i, we do so in a stochastic way: $p(si=1)=11+\exp(-\Delta E/T)$, and $p(si=0)=1-p(si=1)$. Here, $\Delta E$ is the energy gap, i.e. the energy when the unit is off, minus the energy when the unit is on. T is the temperature. We can run our system with any temperature that we like, but the most commonly used temperatures are 0 and 1. When we want to explore the configurations of a Boltzmann Machine, we initialize it in some starting configuration, and then repeatedly choose a unit at random, and pick a new state for it, using the probability formula described above. Consider two small Boltzmann Machines with 10 units, with the same weights, but with different temperatures. One, the "cold" one, has temperature 0. The other, the "warm" one, has temperature 1. We run both of them for 1000 iterations (updates), as described above, and then we look at the configuration that we end up with after those 1000 updates. Which of the following statements are true? (Note that an "energy minimum" is what could also reasonably be called a "local energy minimum")

- CHECKED (b/c cold one isn't really stochastic) - The cold one is more likely to end in an energy minimum than the warm one.
- CHECKED (cold one can only take values 0 or 1) - For the warm one, P(si=1) can be any value between 0 and 1, depending on the weights.
- CHECKED - The warm one could end up in a configuration that's not an energy minimum.

- NOT (the exponent of e can only be +/-inf) - For the cold one, P(si=1) can be any value between 0 and 1, depending on the weights.
- CHECKED (there are 10! possible states so 1000 iters wouldn't cover them all) - If the cold one is exponentially unfortunate, it could end up in a configuration that's not an energy minimum.
- NOT (INCORRECT, b/c warm will maintain state more often due to stocasticity) - If the weights are small, then over the course of those 1000 updates, the warm one is likely to have encountered more different configurations than the cold one.

4. The Boltzmann Machine shown below has two visible units V1 and V2. There is a connection between the two, and both units have a bias. Let W12=−2, b1=1, and b2=1. What is P(V1=1,V2=0)? Write your answer with at least 3 digits after the decimal point.

- 2.718 (INCORRECT, need to normalize wrt other configurations, should be 0.36_)

5. The figure below shows a Hopfield network with five binary threshold units: a, b, c, d, and e. The network has many connections, but no biases. Let Wac=Wbc=1, Wce=Wcd=2, Wbe=−3, Wad=−2, and Wde=3. What is the configuration that has the lowest energy? What is the configuration that has the second lowest energy (considering all configurations, not just those that are energy minima)? A configuration consists of a state for each unit. Write "1" for a unit that's on, and "0" for a unit that's off. To describe a configuration, first write the state of unit a, then the state of unit b, etc. For example, if you want to describe the configuration where units a and d are on and the other units are off, then write 10010. For this question you have to describe two configurations, and write them with a comma in between. For example, if you think that the lowest energy configuration is the one where only units a and d are on, and that the second lowest energy configuration is the one where only units b, d, and e are on, then you should write this: 10010, 01011

- A: "00111, 10111" (with a space!)

## Lecture 12a: The Boltzmann Machine learning algorithm

- Unsupervised learning, learn an input vector "although it might make more sense to think of them as output vectors"
- Goal: maximize the product of the probabilities that the Boltzmann machine assigns to the binary vectors in the training set.
- What exactly is being learned by the Boltzmann Machine learning algorithm? A: Parameters that define a distribution over the visible vectors.
- Why the learning could be difficult
  - Consider a chain of units with visible units at the ends
    - v -> w1 -> h -> w2 -> h -> w3 -> h -> w4 -> h -> w5 -> v
  - If the training set consists of (1,0) and (0,1) we want the product of all the weights to be negative (because need the change the sign from 1 to 0/-1 or from 0/-1 to 1 when traversing from visible to visible)
  - to know how to change w1 or w5 we must know w3
- 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.
  - del log(p(v)) / del w_ij = <s_i,s_j>_v - <s_i,s_j>_model ... using <> now to denote *expected value*
  - derivative of log P of one training vector, v, under the model: del log(p(v)) / del w_ij
  - expected value of product of states, i and j, when the network settles at thermal equilibrium when v is clamped on visible units: <s_i,s_j>_v
    - i.e. how often are i and j on together, when v is clamped
  - expected value of product of states at thermal equilibrium with no clamping: <s_i,s_j>_model
  - learning rule is then: $\Delta w_{ij} \propto$ <s_i,s_j>_data - <s_i,s_j>_model
  - expected product of the activities averaged over all visible values in the training set: <s_i,s_j>_data
  - 1st term: increase the weights in proportion to the product of activities units have when you're presenting data (simplest form of Hebian learning rule, Donald Heb)

    - 2nd term: reducing weights in proportion to how often two units are on together when sampling from the model's distribution (w/out this adjustment/controlling term, the learning blows up and all weights become very positive)
    - 1st term: storage term for Hopfield Net
    - 2nd term: term for getting rid of suprious minima (i.e. how much unlearning to do)
- Why is the derivative so simple?
    - The probability of a global configuration *at thermal equilibrium* is an exponential function of its energy: exp(-E)
        - 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:
        - 
            - del E / del w_ij = s_i * s_j
    - The process of settling to thermal equilibrium propagates information about the weights.
        - We don't need backprop.
        - We do need 2 stages: we need to settle with the data, and we need to settle with no data, but notice that the network is behaving approx. in the same way btw those 2 phases, the unit deep within the network is doing the same thing, just with different boundary conditions
- Why do we need the negative phase?
    - recall: p(v) = sum_h[exp(-E(v,h))] / sum_u[ sum_g[ exp(-E(u,g)) ] ]
    - numerator: the positive phase (first term) finds hidden configurations that work well with v and lowers their energies
    - denominator: the negative phase (second term) finds the joint configurations that are the best competitors and raises their energies
    - First term is making the top line big, and second term is making the bottom line small
- 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.
        a. Update the *hidden units* one at a time until the network reaches thermal equilibrium at a temperature of 1
        b. Sample <s_i,s_j> for every connected pair of units
        c. Repeat for all data vectors in the training set and average
    - *Negative phase*: Set all the units (h and v) to random binary states.
        a. Update *all* the units one at a time until the network reaches thermal equilibrium at a temperature of 1 (same as in Positive Phase)
        b. Sample <s_i,s_j> for every connected pair of units.
        c. Repeat many times (how many?) and average to get good estimates.
    - Especially in the Negative Phase expect the energy landscape to have many minima that are fairly separated and have about the same energy
    - This is b/c we're going to be using Boltzmann Machines to do things like model a set of images, and we expect there to be reasonable images (w/ low energy, small fraction of the space) and unreasonable images (w/ much higher E, high fraction of the space). If have multiple modes, it's very unclear how many times this process needs to be repeated to sample all those modes.

## Lecture 12b: More efficient ways to get the statistics

- ADVANCED MATERIAL: NOT ON QUIZZES OR FINAL TEST
- A better way of collecting the statistics
    - If we start [the weights] from a random state [in either phase, positive or negative], 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)

    i. 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 [FWC - keeping weights fixed] 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.
    ii. Negative phase: Keep a set of "fantasy particles". *Each particle has a value that is a global configuration.*
    - Sequentially update all the units [FWC - keeping weights fixed] in each fantasy particle a few times.
    - For every connected pair of units, average s_i,s_j over all the fantasy particles.
    iii. Learn: $\Delta w_{ij} \propto$ <s_i,s_j>_data - <s_i,s_j>_model
- FWC - this is all very much like negative sampling again, positive phase (real data), negative phase (fake/negative data--or at least a higher probability of being so)
- One way to tell if you've learned a good model is after learning, remove all the input, and just generate samples. Run the Markov Chain for a long time until it's burned in, and then look at the samples you get
- A puzzle
    - Why can we estimate the [MNIST data] "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 [FWC - as in "negative sampling").**
        - We cannot analyze 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**. [FWC - b/c the energy surface is defined as the difference between the the <positive_data>=<si,sj>_data and <fantasy particles>=<random data>=<si,sj>_model]
        - This makes the fantasies rush around hyperactively. [FWC - b/c once they end up raising the energy surface in one area, <si,sj>_model makes them more likely to appear in another area with lower energy]
        - 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
    - **see picture on slide 19 of lec12.pdf**
    - **If a mode (aka local minima) has more fantasy particles than data, the energy surface is raised** [FWC - local minima is pushed upwards so that it's not so much of a minima, like pushing up a blanket holding balls] 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.)

## Lecture 12c: Restricted Boltzmann Machines

- Also see this nice RBM tutorial at DeepLearning4J as well as this follow-up: Understanding RBMs
    - Reconstruction does something different from regression, which estimates a continous value based on many inputs, and different from classification, which makes guesses about which discrete label to apply to a given input example. Reconstruction is making guesses about the probability distribution of the original input; i.e. the values of many varied points at once. This is known as generative learning, which must be distinguished from the so-called discriminative learning performed by classification
    - The question the RBM is asking itself on the forward pass is: Given these pixels, should my weights send a stronger signal to the elephant node or the dog node? And the

question the RBM asks on the backward pass is: Given an elephant, which distribution of pixels should I expect? That's joint probability: the simultaneous probability of `x` given `a` and of `a` given `x`, expressed as the shared weights between the two layers of the RBM. The process of learning reconstructions is, in a sense, learning which groups of pixels tend to co-occur for a given set of images.
    - While RBMs have many uses, proper initialization of weights to facilitate later learning and classification is one of their chief advantages. In a sense, they accomplish something similar to backpropagation: they push weights to model data well. You could say that pre-training and backprop are substitutable means to the same end.
    - A continuous restricted Boltzmann machine is a form of RBM that accepts continuous input (i.e. numbers cut finer than integers) via a different type of contrastive divergence sampling. This allows the CRBM to handle things like image pixels or word-count vectors that are normalized to decimals between zero and one.
        - **Gaussian transformations do not work well on RBMs' hidden layers. The rectified-linear-unit transformations used instead are capable of representing more features than binary transformations**, which we employ on deep-belief nets.
    - **It should be noted that RBMs do not produce the most stable, consistent results of all shallow, feedforward networks. In many situations, a dense-layer autoencoder works better.** Indeed, the industry is moving toward tools such as **variational autoencoders**.
- Restrict the connectivity to make inference and learning easier.
    - Only one layer of hidden units.
    - No connections between hidden units.
    - I.e. a bipartitie graph
- 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 <v_i*h_j>_v (and recall that <> is the expectation function)
    - **p(h_j=1) = 1 / (1 + exp(-b_j-sum_i[v_i * w_ij]))** ... the logistic function
    - **FWC - per programming assignment 4 this function is how you compute both visible_state_to_hidden_probabilities.m and hidden_state_to_visible_probabilities.m (albeit with another dimension for the number of configurations/particles that are being handled in parallel)**
        - also see slide 21 of lec12_boltzmann_machines.pdf
- PCD: An efficient mini-batch learning procedure for Restricted Boltzmann Machines (Tieleman, 2008)
    - PCD == Persistent Contrastive Divergence
    - Positive phase: Clamp a datavector on the visible units.
        - Compute the exact value of <v_i*h_j> for all pairs of a visible and a hidden unit.
        - For every connected pair of units, average <v_i*h_j> 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 [FWC - you can only do *parallel* updates in an **R**BM b/c the it uses a bipartite graph)
        - For every connected pair of units, average v_i->h_j over all the fantasy particles.
    - Allows RBMs to build good density models of sets of binary vectors (i.e. to model their probability density functions)
- A picture of an inefficient version of the Boltzmann machine learning algorithm for an RBM
    - Use times now not to denote weight updates but to denote **steps in a Markov chain**
    - t=0: start by clamping a datavector on the visible units
        - update all hidden units (in parallel) given visible by computing <v_i*h_j>_0
    - t=1: update all the visible units (in parallel) given the new hidden units (a "reconstruction" of v)
        - update hidden again: <v_i*h_j>_1
    - t=2... repeat until
    - t=infinity: thermal equilibrium: <v_i*h_j>_infinity (a fantasy)
    - then the learning rule is: Δw_ij = epsilon(<v_i*h_j*>_0 - <v_i*h_j>_infinity)

    - Of course the problem is that we have to run this for many repetitions to reach thermal equilibrium else the learning may go wrong.
        - Actually this last statement is misleading. It turns out to still work, even if we only run the learning for a short time.
- **Contrastive divergence: A very surprising short-cut**
    - [FWC - like negative sampling again? just change the input so that it's a little wrong, the reconstruction, not the global "wrong"]
    - stop after computing <v_i*h_j>_1
    - Instead of measuring the statistics at equilibrium, we **measure after one full update of the Markov chain**
    - Δw_ij = epsilon(<v_i*h_j>_0 - <v_i*h_j>_1) ... this is CD1 (constrastive divergence, 1 step)
    - Clearly this is not doing maximum likelihood learning b/c the term we're using for the negative statistics is wrong
    - This is not following the gradient of the log likelihood. But it works well.
    - from programming assignment 4: "we'll sample a binary state for the hidden units conditional on the data; we'll sample a binary state for the visible units conditional on that binary hidden state (this is sometimes called the 'reconstruction' for the visible units); and we'll sample a binary state for the hidden units conditional on that binary visible 'reconstruction' state"
- 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--towards things it likes b/c of its initial weights, rather than b/c of the data
        - 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**
        - **FWC - the weights are bad b/c they haven't been learned yet early in the procedure**
    - All we need to do is lower the probability of the confabulations/reconstructions (the former is a psych term) 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.
- 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.
        - [FWC - because they won't be ever be affected by a negative phase b/c we're only going 1 step which doesn't move very far from positive phase / real datapoints]
        - These low energy holes cause the normalization term (denominator) to be big and we cannot sense them if we use the shortcut.
        - Persistent particles [weight configurations] 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 [FWC - since the weights will have meaning now, we'll be getting into regions of the global state space we haven't visited before so that they can be "disproven"]
        - Once the weights have grown more we use CD10.
    - By increasing the number of steps as the weights grow, we can keep the learning working well, even as the mixing rate of the Markov chain is going down
- From programming assignment 4
    - If you go through the math (either on your own on with your fellow students on the forum), you'll see that *sampling the hidden state that results from the "reconstruction" visible state is useless: it does not change the expected value of the gradient estimate that CD-1 produces; it only increases its variance. More variance means that we have to use a smaller learning rate, and that means that it'll learn more slowly; in other words, we don't want more variance, especially if it doesn't give us anything pleasant to compensate for that slower learning. Let's modify the CD-1 implementation to simply no longer do that*

*sampling at the hidden state that results from the "reconstruction" visible state*. **Instead of a sampled state, we'll simply use the conditional probabilities**.

- We want to train our RBM on the handwritten digit data that we used in programming assignment 3, but that presents a **problem: that data is not binary (it's pixel intensities between 0 and 1), but our RBM is designed for binary data**. We'll treat each training data case as a distribution over binary data vectors. A product of independent Bernoulli-distributed random variables, if you like mathematical descriptions. What it means in practice is that every time we have a real-valued data case, we **turn it into a binary one by sampling a state for each visible unit**, where we treat the real-valued pixel intensity as the probability of the unit turning on. Let's add this line of code as the new first line of the cd1 function: visible_data = sample_bernoulli(visible_data);
- Part 3: Using the RBM as part of a feedforward network: [FWC - **Constructive Distraction**]
  - Here's the plan: *we're going to train an RBM (using CD-1), and then we're going to make the weights of that RBM into the weights from the input layer to the hidden layer, in the deterministic feed-forward network* that we used for PA3 . We're not going to tell the RBM that that's how it's going to end up being used, but a variety of practical and theoretical findings over the past several years have shown that this is a reasonable thing to do anyway. The lectures explain this in more detail. This brings up an interesting contrast with PA3. In PA3, we tried to reduce overfitting by learning less (early stopping, fewer hidden units, etc). This approach with the RBM, on the other hand, reduces overfitting by learning more: **the RBM part is being trained unsupervised, so it's working to discover a lot of relevant regularity in the distribution of the input images, and that learning distracts the model from excessively focusing on the digit class labels. This is much more constructive distraction**: instead of early-stopping the model after only a little bit of learning, we instead give the model something much more meaningful to do. In any case, it works great for regularization, as well as training speed
  - In PA3, *we did a very careful job of selecting the right number of training iterations, the right number of hidden units, and the right weight decay. For PA4, we don't need to do all that: although it might still help a little, the unsupervised training of the RBM basically provides all the regularization we need*
- Of course, you can do much more. For example, explore what number of hidden units works best, and you'll see that that number is indeed much larger than it was on PA3. Or use your PA3 code to properly train the feedforward NN after its RBM initialization. Or add some more hidden layers. Or... creatively combine everything else that you're learning in this course, to see how much this **RBM-based unsupervised pre-training** can do.

## Lecture 12d: An example of Contrastive Divergence Learning

- How to learn a set of features that are good for reconstructing images of the digit '2'
  i. real data (reality): 16x16 pixel image
  ii. -> 50 binary neurons that learn features
  - *increment* weights between pixels and an active (real) feature, v_i0 (i.e. increment w_ij0 if v_i0 and h_j0 are both 'on') when the network is looking at data, which will lower the energy of the data (and any hidden pattern that went with it)

  iii. -> reconstructed data (better than reality): 16x16 pixel image
  iv. -> 50 binary neurons that learn features
  - *decrement* weights between an active (reconstructed) pixel, v_i1, and an active feature, h_j1 (which will raise the energy of a reconstruction)
  - Near the beginning of learning, when the weights are random, the reconstruction will almost surely have lower weights than the real data b/c the reconstruction is what the network likes to reproduce on the visible units, given the hidden pattern of activity
- The weights of the 50 feature detectors
  - Initially, after only a few steps, global feature vectors are learned
  - Specialization happens eventually (e.g. features to recognize the top of a '2')
- How well can we reconstruct digit images from the binary feature activations?
  - Compare reconstruction of a '2' which yields something that looks like a '2' vs. reconstruction of a '3' which also yields something that looks like a garbled '2'. 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.
  - *Feature detectors* can pick up on long-range irregularities (e.g. pixel 1 on while 237 is off)
  - See slide 41 of lec12.pdf
    - 2nd row, 6th column
    - shows pairs of side-by-side pixels that cannot be both on or both off
    - sorta like the least significant digit in a binary number that alternates between 0 and 1

## Lecture 12e: RBMs for collaborative filtering

- "Not the type of thing that anybody at the time thought an RBM could deal with. There's an important trick to get an RBM to deal" with the sparsity of the data.
- 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 $1,000,000
- Lets (start) by using 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.
  - Convert each user into a <user-feature-vector> and each movie into a <movie-feature-vector> and from those try to predict the rating
  - Obvious way to do this is to put in a big hidden layer, which was tried, and no better than a very simple method: dot product of <ufv> and <mfv> (not even a softmax).
    - This is **exactly equivalent to doing matrix factorization**: <users_x_features_matrix> * <features_x_movies_matrix>
    - Matrix factorization model is most commonly used model for collaborative filtering like this, and it works pretty well.
      - FWC - update 1/23/16 - Matrix Factorization with Tensorflow
- ********** **An RBM alternative to matrix factorization** **********
  - 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.
- How to avoid dealing with all those missing ratings
  - We don't want an RBM to have to deal with 18,000 visible units, but with only a few missing values
  - 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 [collaborative filtering] 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).

## Lecture 12 Quiz

1. The Boltzmann Machine learning algorithm involves computing two expectations
2. <s_i,s_j>_data: Expected value of sisj at equilibrium when the visible units are fixed to be the data.
3. <s_i,s_j>_model: Expected value of sisj at equilibrium when the visible units are not fixed. When applied to a general Boltzmann Machine (not a Restricted one), this is an approximate learning algorithm because

- CHECKED [lec12.pdf, p. 22] - There is no efficient way to compute the first expectation exactly.
- CHECKED (all correct: Computing <s_i*sj*>_*data and* <*s_i*sj>_model are hard in general. They usually involves sampling from the model conditioned on the data.) - There is no efficient way to compute the second expectation exactly.
- UNCHECKED - The first expectation can be computed exactly, but the second one cannot be.
- UNCHECKED - The first expectation cannot be computed exactly, but the second one can be.

2. Throughout the lecture, when talking about Boltzmann Machines, why do we talk in terms of computing the expected value of sisj and not the value of sisj ?

- SHOULD'VE BEEN CHECKED - It does not make sense to talk in terms of a unique value of sisj because si and sj are random variables and the Boltzmann Machine defines a probability distribution over them.
- It is not possible to compute the exact value no matter how much computation time is provided. So all we can do is compute an approximation.
- It is possible to compute the exact value but it is computationally inefficient.
- CHECKED (incorrect) - The expectation only refers to an average over all training cases.

3. When learning an RBM, we decrease the energy of data particles and increase the energy of fantasy particles [FWC - see slide 26 of lec12.pdf]. Brian insists that the latter is not needed. He claims that it is should be sufficient to just decrease the energy of data particles and the energy of all other regions of state space would have increased relatively. This would also save us the trouble of sampling from the model distribution. What is wrong with this intuition ?

- There is nothing wrong with the intuition. This method is an alternative way of learning a Boltzmann Machine.
- Since total energy is constant, some particles must loose energy for others to gain energy.
- CHECKED (The network might update its weights to lower the energy of large regions of space surrounding the data particles and these regions and the decrease could be as large as possible. Another way to see this is to look at the weight update equation and notice that the gradient would never go to zero unless there are negative particles.) - The model could decrease the energy of data particles in ways such that the energy of negative particles also gets decreased. If this happens there will be no net learning and energy of all particles will keep going down without bounds.
- The sum of all updates must be zero so we need to increase the energy of negative particles to balance things out.

4. Restricted Boltzmann Machines are easier to learn than Boltzmann Machines with arbitrary connectivity. Which of the following is a contributing factor ?

- CHECKED (This makes it possible to update all hidden units in parallel given the visible units (and vice-versa). Moreover, only one such update gives the exact value of the expectation that is being computing.) - In RBMs, there are no connections among hidden units or among visible units.
- It is possible to run a persistent Markov chain in RBMs but not in general BMs.
- RBMs are more powerful models, i.e., they can model more probability distributions than general BMs.
- The energy of any configuration of an RBM is a linear function of its states. This is not true for a general BM.

5. **PCD is a better algorithm than CD1 when it comes to training a good generative model of the data.** This means that samples drawn from a freely running Boltzmann Machine which

was trained with PCD (after enough time) are likely to look more realistic than those drawn from the same model trained with CD1. Why does this happen ?

- In PCD, the persistent Markov chain can remember the state of the positive particles across mini-batches and show them when sampling. However, CD1 resets the Markov chain in each update so it cannot retain information about the data for a long time.
- In PCD, only a single Markov chain is used throughout learning, whereas CD1 starts a new one in each update. Therefore, PCD is a more consistent algorithm.
- In PCD, many Markov chains are used throughout learning, whereas CD1 uses only one. Therefore, samples from PCD are an average of samples from several models. Since model averaging helps, PCD generates better samples.
- CHECKED - **In PCD, the persistent Markov chain explores different regions of the state space. However, CD1 lets the Markov chain run for only one step. So CD1 cannot explore the space of possibilities much and can miss out on increasing the energy of some states which ought to be improbable. These states might be reached when running the machine for a long time leading to unrealistic samples.**

6. It's time for some math now! In RBMs, the energy of any configuration is a linear function of the state. E(v,h) = -∑_i[a_i*v_i*] - ∑_*j*[*b_j*h_j] - ∑_ij[v_i*h_j*w_ij] and this eventually leads to Δw_ij ∝ <v_i*h_j*>_*data* - <*v_i*h_j>_model. If the energy was non-linear, such as E(v,h) = -∑_i[a_i*f(v_i)*] - ∑_*j*[*b_j*g(h_j)] - ∑_ij[f(v_i)*g(h_j)*w_ij] for some non-linear functions f and g, which of the following would be true.

- ΔWij∝⟨f(vi)⟩data⟨g(hj)⟩data−⟨f(vi)⟩model⟨g(hj)⟩model
- CHECKED (b/c the energy function is still linear in the weights, which makes the derivative easy--just remove the w_ij) - Δw_ij ∝ <f(v_i),g(h_j)>_data - <f(v_i),g(h_j)>_model
- ΔWij∝⟨vihj⟩data−⟨vihj⟩model
- ΔWij∝f(⟨vi⟩data)g(⟨hj⟩data)−f(⟨vi⟩model)g(⟨hj⟩model)
- above is CORRECT and here's why
  - p(v) = exp(-E(v,h)) / Z
  - ⇒ log(p(v)) = -E(v,h) - log(Z)
  - ⇒ ∂log(p(v)) / ∂w_ij = f(vi)g(hj) - ∑_{v′,h′}[P(vi′,hj′)*f(vi′)*g(hj′)]
  - Averaging over all data points,
  - ∂log(p(v)) / ∂w_ij = <f(vi)*g(hj)>_data - <f(vi)*g(hj)>_model
  - Δw_ij ∝ ∂log(p(v)) / ∂w_ij
  - ⇒ Δw_ij ∝ <f(vi)*g(hj)>_data - <f(vi)*g(hj)>_model

7. In RBMs, the energy of any configuration is a linear function of the state. E(v,h) = −∑iaivi − ∑jbjhj − ∑i,jvihjWij and this eventually leads to P(hj=1|v) = 1/(1+exp(−∑iWijvi−bj)). If the energy was non-linear, such as E(v,h) = −∑iaif(vi) − ∑jbjg(hj) − ∑i,jf(vi)g(hj)Wij for some non-linear functions f and g, which of the following would be true.

- P(hj=1|v)=1/(1+exp(−∑iWijf(vi)−bj))
- SHOULD'VE BEEN CHECKED - P(hj=1|v)=1/(1+exp((g(0)−g(1))(∑iWijf(vi)+bj)))
- P(hj=1|v)=1/(1+exp(−∑iWijvi−bj))
- CHECKED (incorrect) - None of these is correct.

8. A Boltzmann Machine is different from a Feed Forward Neural Network in the sense that:

- CHECKED - **A Boltzmann Machine defines a probability distribution over the data, but a Neural Net defines a deterministic transformation of the data.**
- UNCHECKED - The state of a hidden unit in a Boltzmann Machine is a deterministic function of the inputs and is hard to compute exactly, but in a Neural Net it is easy to compute just by doing a forward pass.
- UNCHECKED - Boltzmann Machines do not have hidden units but Neural Nets do.
- CHECKED - **The state of a hidden unit in a Boltzmann Machine is a random variable [FWC - b/c they are updated in response to the visible units (given the function defined by the weights), which are also random variables], but in a Neural Net it is a deterministic function of the inputs.**

## Lecture 13a: The ups and downs of backpropagation

- "In machine learning, in the 90s, people thought backpropagation had been supplanted by support vector machines."
- 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 (initialized with too small of weights that died out)
- **A spectrum of machine learning tasks**
  - Typical Statistics ---------- Artificial Intelligence
    - Low-dimensional data (e.g. < 100D)
    - 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 thinking that noise is really structure.
      - *Not ideal for non-Bayesian neural nets*. Try SVM or GP.
  - AI
    - High-dim (> 100D)
    - The noise is not the main problem
    - There is a huge amount of structure in the data, but it's 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.
      - The natural thing to do is to hand design the representation, but actually the better thing to do is to let backpropagation figure it out via multiple layers. and use a lot of computation power to let it decide what the representation should be.
- Why Support Vector Machines were never a good bet for Artificial Intelligence tasks that need good representations (2 views)
  - View 1: SVM's are just a clever reincarnation of Perceptrons.
    - very efficient/clever way of fitting the weights that controls overfitting (max separating margin)
  - View 2: SVM's are just a clever reincarnation of Perceptrons.
    - use each input vec in training set to define a non-adaptive "pheature"
    - the global match btw a
  - both views are essentially the same: in both cases SVMs are using non-adaptive features and 1 layer of adaptive weights
  - can't learn multiple layers of representation with a SVM
- Historical document (1995) from AT&T Adaptive Systems Research Dept., Bell Labs
  - Jackel and Vapnik were both wrong
  - the limitation wasn't that we didn't have good enough theory or that they were endlessly hopeless, it was that we didn't have big enough computers and big enough datasets (practical limitation, not theoretical)

## Lecture 13b, Belief Nets

- "one of the reasons I abandoned backprop in the 90s was because it required too many labels, I was also influenced by the fact that people learn with very few explicit labels"
- "I didn't want to abandon the advantages of doing GD learning to learn a whole bunch of weights, so the issue was: **was there another objective function we could do GD in? the obvious place to look was generative models [FWC - e.g. GANs?] where the goal is to match the input data rather than predicting a label**"
- Graphical Models (GM): combine discrete graph structures for representing how variable depend on each other w/ real valued computations that infer one variable based on values of other vars
- BMs undirected GMs; directed GMs: "sigmoid belief nets"
- Second problem: for deep nets learning time doesn't scale well (b/c of poor initialization)
  - back prop can get stuck in poor local optima
  - for deep nets the local optima you get stuck in are due to poor weight init

    - **possible to retreat to simpler models that allow convex opt "but i don't think that's a good idea. mathematicians like to do that because they can prove things, but in practice you're just running away from the complexity of real data"**
- 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 (hard), first learn computer graphics (easy).
    - The learning objective for a generative model:
        - Maximise p(x) [probability of generating the input] 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
    - When Hinton was a graduate student you were regarded as "stupid" if you thought probability had anything to do with AI. It would just muck up your deterministic/discrete equations.
    - Old view: “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)
    - New view: “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
    - Probability eventually found its way into AI by way of graphical models.
    - 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! "you can actually prove this is a bad bet"
    - 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 (generally the leaves) 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.
            - I.e. Decide both which nodes are affected by which other nodes and decide thes strengths of those effects.
- Graphical Models vs. 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, e.g. CNN architecture).
        - 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 (RBM), it is easy to learn a Boltzmann machine.
        - But then we only have one hidden layer (and we've given up on the benefits of deep networks)
    - Causal [b/c it's a DAG, no symmetric connections]: We connect binary stochastic neurons in a *directed acyclic graph* to get a Sigmoid Belief Net (Neal 1992).
        - Neal showed these are slightly easier to learn then BMs
        - in a Causal model, unlike a BM, it's easy to generate samples
- The Math of Sigmoid Belief Nets

## Lecture 13c: Learning Sigmoid Belief Nets

- Good news: don't need 2 different phases--just need (what would be in a Boltzmann Machine) the positive phase because SBNs are "locally normalized models" so we don't have to deal w/ a partition function or its derivatives
- Learning Sigmoid Belief Nets
    - It is easy (because it's a causal model; i.e. top-down) 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.
        - B/c the number of possible patterns of hidden causes is exponential in the number of hidden nodes.
    - It is hard to even get a sample from the posterior (which we need if we're going to use SGD).
    - So how can we learn sigmoid belief nets that have millions of parameters?
- The learning rule for sigmoid belief nets
    - Learning is easy if (a big 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.
        - Learning rule: $\Delta w_{ji} = \varepsilon * s_j ( s_i - p_i )$
        - $s_j$ : state of parent
        - $s_i$ : state of child
        - $p_i$ : probability that states of i's parents would turn it on
    - To summarize: if we have an assignment of binary states to all of the hidden nodes, then it's easy to do maximum likelihood learning (in our typical stochastic way where we sample from the posterior, and then we update the weights based on that sample--and we average that update over a mini batch of samples)
- Explaining away (Judea Pearl)
    - **The reason it's hard to get an unbiased sample from the posterior over the hidden nodes, given an observed (visible) data vector at the leaf nodes, is a phenomenon called "explaining away."**
    - Even if two hidden causes are independent in the prior (in the model; the connectivity of the network), 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.
        - if "house jumps" C can be caused by "truck hits house" A or "earthquake" B .... and we observe C and A .... then we can "explain away" B

    - Note the picture in the lecture shows -10 biases going into "truck hits house" and "earthquake" which means they're both not very likely = 1/(1+exp(-10))
    - but when either of those occurs there's a +20 weight that cancels out the -20 bias on "house jumps" so they each have a probability of 0.5 = 1/(1+exp(-20+20))
    - The posterior actually looks something like this. There's 4 possible patterns of hidden causes given that the house jumped. Two are extremely unlikely, namely that a truck hit the house and there was an earthquake or that neither of those things happened (again, given that the house jumped). The other 2 combinations are equally probable--XOR.
        - p(0,0)=0.0001
        - p(0,1)=0.4999
        - p(1,0)=0.4999
        - p(1,1)=0.0001
    - We have 2 likely causes of something that are just the opposites of each other. That's "explaining away."
- Why it's hard to learn sigmoid belief nets one layer at a time
    - To learn W (the last/bottom hidden layer to input data weights), we need to sample from the posterior distribution in the first hidden layer.
        - Problem 1: The posterior (bottom-up from data) is not factorial (the nodes in the bottom hidden layer aren't independent) because of "explaining away". But because we have higher layers of hidden variables, they aren't even independent in the prior (top-down).
        - 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. It's 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?
    - There in fact is a guarantee that something will improve.
        - It's not the log probability that the model will generate the data, but it's related--a lower bound on that log--and by pushing up the lower bound, we can push up the log prob.

## Lecture 13d: The wake-sleep algorithm

- WS should not be confused with Boltzmann Machines. They have 2 phases, the positive and negative phase, that could plausibly related to wake and sleep, but the Wake-Sleep Algorithm is a very different kind of learning, namely because it is for directed graphs.
- Variational Methods
    - b/c it's hard to compute the real posterior distribution, we'll compute some cheap approximation to it, and then we'll do maximum liklihood learning anyway. that is we'll apply the learning rule that would be correct if we'd gotten a sample from the true posterior and hope that it works, even though we haven't
    - you could reasonably expect this to be a complete disaster, but actually the learning comes to the rescue
    - when computing the true posterior there are actually 2 terms driving the weights
        - 1 term is driving them to get a better model of the data, that is to make the SBN more likely to generate the observed data in the training set
        - another term is driving the weights towards sets of weights for which the approx. posterior is a good fit for the real posterior. it does this by manipulating the real

posterior to try to fit the approx posterior. b.c of this effect that variational learning of these models works quite nicely

- 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.
    - in other words, we're going to assume that given the data, the units in each hidden layer are independent of one another [FWC - because "factorizes" => can be multiplied together w/ no correlation term] as they are in an RBM (but in an RBM this assumption is correct while in a SBM it's wrong)
- 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 -> 0.3 0.6 0.8
    - probability that the hidden units have state 1,0,1 if the distribution is factorial. p(1, 0, 1) = 0.3 × (1−0.6) × 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). [FWC - due to full connectivity; all nodes connected to all other nodes]
  - A factorial distribution only has N degrees of freedom ("it's a much simpler beast").
- The wake-sleep algorithm (Hinton et. al. 1995)
  - 2 sets of weights
    - *recognition weights* R_i - forward pass weights - from input to max hidden layer (bottom-up)
    - *generative weights* w_i - backward pass - to "generate" input, from hiddens to input (top-down)
  - Wake phase: Use *recognition weights* R_i (one set for each layer) to perform a bottom-up pass.
    - Train the generative weights to reconstruct activities in each layer from the layer above.
    - Drive the system in the forward pass w/ the recognition weights but do maximum likelihood to learn the generative weights
    - At each hidden layer, make a stochastic decision, for each binary unit independently, about whether it should be on or off.
      - The forward pass gets us stochastic binary states for all of the hidden units.
      - Once we have those, we treat them as if they were a sample from the true posterior distribution, and do maximal likelihood learning
  - Sleep phase: Use generative weights to generate samples from the model.
    - Start with random activities of the top hidden layer, and use generative weights to figure out each layer below. "generating states for each layer at a time" (i.e. using the generative model "correctly"--it's how the generative model says you aught to generate data)
    - having used the generative weights to generate an unbiased sample, you then say, lets see if we can recover the hidden states from the data (recover the hidden states from layer h2 from layer h1)
    - train the recognition weights to try to recover the hidden states that actually generated the states below
    - 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. (Hinton thinks it has too many problems though and that we'll find better algorithms)
- Mode averaging [FWC - this is like my blurriness idea for predicting the next frame in a video--you don't want to average over all possible future frames (the true average of which could never happen)--you want to pick one possible future frame (i.e. sharpen the blurriness)]
    - 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) [where having a 1 in either location occurs w/ P=1e-9 or some really small number]
        - 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 (a quarter on each): very improbable (1e-18 for 1,1 and 0,0 is improbable because the visible unit is on--the "house moved") hidden configurations.
    - **It's 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 (if you're forced to have a factorial model)
    - In variational learning we're manipulating the true posterior (FWC - i.e. pick one mode and call that the mean of the distribution) to make it fit the approximation we're using. Normally in learning, we're manipulating an approximation (FWC - i.e. select the mean of all the modes) to fit the true thing.

## Lecture 13 Quiz

1. This quiz is going to take you through the details of Sigmoid Belief Networks (SBNs). The most relevant videos are the second video ("Belief Nets", especially from 11:44) and third video ("Learning sigmoid belief nets") of lecture 13. We'll be working with this network: two hidden units `h1` and `h2` connected by `w1` and `w2` to one visible unit `v`. The network has no biases (or equivalently, the biases are always zero), so it has only two parameters: w1 (the weight on the connection from h1 to v) and w2 (the weight on the connection from h2 to v). Remember, the units in an SBN are all binary, and the logistic function (also known as the sigmoid function) figures prominently in the definition of SBNs. **These binary units, with their logistic/sigmoid probability function, are in a sense the stochastic equivalent of the deterministic logistic hidden units that we've seen often in earlier lectures.** Let's start with w1=−6.90675478 and w2=0.40546511. These numbers were chosen to ensure that the answer to many questions is a very simple answer, which might make it easier to understand more of what's going on. Let's also pick a complete configuration to focus on: h1=0,h2=1,v=1 (we'll call that configuration C011). Ready to Begin? (Please select a response. This question is reflective and selecting a certain answer will not affect your grade.)
2. What is P(v=1|h1=0,h2=1)? Write your answer with four digits after the decimal point. Hint: the last three of those four digits are zeros. (If you're lost on this question, then I strongly recommend that you do whatever you need to do to figure it out, before proceeding with the rest of this quiz.)

- A: sigmoid(1 / (1 + exp(-0.40546511)) = 0.6000
- Correct: Pretend that this is a feed-forward neural network with two hidden units and a logistic output neuron. You're now calculating the output of the network given that the hidden units have taken on the values h1=0 and h2=1.

3. What is the probability of that full configuration, i.e. P(h1=0,h2=1,v=1), which we called P(C011)? Write your answer with four digits after the decimal point. Hint: it's less than a half, and the last two of those four digits are zeros.

- A: P(h1=0,h2=1,v=1) = P(v=1 | h1=0,h2=1) * P(h1=0,h2=1) = 0.6 * 0.25 = 0.1500
- Correct: We can use the rule of multiplication in order to obtain P(h1=0,h2=1,v=1)=P(v=1|h1=0,h2=1)P(h1=0,h2=1). Question 1 deals with finding P(v1|h1=0,h2=1), now you need to find P(h1=0,h2=1). What does the picture given in the

preamble tell you about the marginal independence of h1 and h2 (when we have not observed v)? Also, remember that h1 and h2 both have 0 total input, and that they are logistic neurons.

4. What is ∂logP(C011)/∂w1? Write your answer with at least three digits after the decimal point, and don't be too surprised if it's a very simple answer.

- A: 0.000
- see slide 17: h1(v-p) = 0*(1-0.6)

5. What is ∂logP(C011)/∂w2? Write your answer with at least three digits after the decimal point, and don't be too surprised if it's a very simple answer.

- A: 0.4000
- see slide 17: h2(v-p) = 1*(1-0.6)

6. *For questions 6 and 7 use: w1=10 and w2=-4 (this text was not in the actual quiz text)* What is P(h2=1|v=1,h1=0)? Give your answer with at least four digits after the decimal point. Hint: it's a fairly small number (and not a round number like for the earlier questions); try to intuitively understand why it's small. Second hint: you might find Bayes' rule useful, but even with that rule, this still requires some thought.

- P(C011) / [P(C011) + P(C001)]
- 0.5455 (INCORRECT due to change in weights that weren't mentioned in the quiz text)
- 0.03472 (CORRECT)
- We can use Bayes' rule to determine: P(h2=1|v=1,h1=0) = P(v=1|h1=0,h2=1)P(h2=1) / (P(v=1|h1=0,h2=1)P(h2=1) + P(v=1|h1=0,h2=0)P(h2=0))

7. What is P(h2=1|v=1,h1=1)? Give your answer with at least four digits after the decimal point. Hint: it's quite different from the answer to the previous question; try to understand why. The fact that those two are different shows that, conditional on the state of the visible units, the hidden units have a strong effect on each other, i.e. they're not independent. That is what we call explaining away, and the earthquake vs. truck network is another example of that.

- P(C111) / [P(C111) + P(C101)]
- 0.5999 (INCORRECT)
- 0.4994 (CORRECT)

### Week 13, Programming assignment 4

- file:///home/fred/Documents/articles/geoff_hinton's_machine_learning_coursera/programming_assigments/assignment_4_week_13/README.txt

## Lecture 14a: Learning layers of features by stacking RBMs

- Anybody sensible would expect if you combined a set of BMs together to make 1 model what you'd get is a multilayer BM (undirected), but you actually get something more like a SBN (directed)
- 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 (only applicable if you do it just right, which we don't do in practice, but at least the proof is reassuring b/c it suggests something sensible is going on).
    - 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 Deep Belief Net (DBN; stacked BMs)
  - see nice picture in lec14_stacking_RBMs.pdf

    - note that if you start the second BM off with the transpose of the weights matrix learned by the first BM then you just get back your initial input (b/c that's what a BM is doing in the first place
    - The lower (directed) layers are like a SBN while the top 2 layers (undirected) form an RBM--i.e. it's not a BM!
- The generative model after learning 3 layers
- To generate data:
    i. Get an equilibrium sample from the top-level RBM by performing alternating Gibbs sampling for a long time. (**The top 2 layers--the top RBM--is defining the prior distribution over the second-to-top-layer**.)
    ii. Starting from second-to-top-layer, 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.
        - What it means "to be factorial" is that the probability of the first 2 units turning on together from the (input vector) v1 distribution (below) is 0.9 * 0.9 = 0.81
        - 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: 0.1, 0.1, 0.9, 0.9
        - Aggregated = 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 = 0.9 * 0.9 * (1-.1) * (1-.1)
        - 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 (which is much smaller)
        - I.e. averaging the two distributions has given us a mixture distribution, which is not factorial
- Why does greedy learning work?
    - Why is it a good idea to learn a RBM and then learn a second RBM that models the patterns of activity in the hidden units of the first one?
    - 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).
    - p(v|h) & p(h|v) - the 2 distributions we use for learning our alternating Markov Chain
    - p(v,h) - if we run that chain long enough, we get a sample from the joint distribution, and so the weights also define the joint distribution (they also define the joint distribution of exp(-E), but for large nets we can't compute that)
    - p(h) & p(v) - if we ignore h, we have the prior distribution over v (and similarly vice versa)
    - We can express the RBM model as: p(v) = ∑_h[ p(h) * p(v|h) ]
    - (this seems like a silly thing to do b/c defining p(h) is just as difficult as defining p(v))
    - If we leave p(v|h) alone and improve p(h), we will improve p(v).
        - a prior over h that fits the aggregated posterior better: the avg of all vectors in the training set of the posterior dist over h
    - 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 (FWC bottom-up?) to the data.
        - use the first/bottom RBM to get the aggregated posterior, then use the second RBM to build a better model of this aggregated posterior than the first (start the second model with the weights from the first upside down / transposed) [FWC - **posteriors are built bottom-up; priors top-down**]
        - **This explains what's happening when we stack up RBMs.**
- 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.
        a. 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.
      b. Do a few iterations of sampling in the top level RBM
        - Then adjust the weights in the top-level RBM using CD.
      c. Do a stochastic top-down pass
        - Then Adjust the bottom-up weights to be good at reconstructing the feature activities in the layer above (up to the input layer of the top level RBM--the second to last hidden layer)
    - This is just the sleep phase.
    - The difference from the standard wake-sleep algorithm is that **the top level RBM acts a much better prior over the top layers than just a layer of units which are assumed to be independent** (as in a SBN)
    - **Rather than generating data by sampling the prior, we take an input training case, work up to the top level RBM and alternate a few times before we generate data**
- The DBN used for modeling the joint distribution of MNIST digits and their labels
    - The first two hidden layers (28x28 pixel image -> 500 hidden units -> 500 hidden units) are learned without using labels
    - The top layer (2000 units) is learned as an RBM for modeling the labels (10-way softmax actually) *concatenated* with the (500) features in the second hidden layer.
    - The weights are then fine-tuned to be a better generative model using contrastive wake-sleep.
    - This is the model from the intro video of this course. Go back and see what happens when you run this model. Good at both recognition and generation!

## Lecture 14b: Discriminative fine-tuning for DBNs

- Instead of fine-tuning to be better at generation (as we did in previous video) we're going to fine-tune it to be better at discriminating classes--which works very well. Major influence in speech recog.
- **Fine-tuning for discrimination** (i.e. for supervised learning)
    - 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**.
            - **[FWC - It would be odd for an *overfit* learned feature to accurately predict labels, other than by extreme chance. First the feature has to get learned (which lowers the chance of it being overfit), then it also would have to be good at predicting labels (which lowers the chance even further). It has to get lucky twice!]**
- 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 (such as w/ CNNs for image recognition; locality = convolution fn)
    - 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 which contain on a few bits of information, not enough to constrain input->output function.
        - 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 [FWC - i.e. shift / linearly transform]. 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. [**FWC - just like in most forecasting we construct good features first, then see how good they are at forecasting**]
    - 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.
        - When computers were much smaller, this was a reasonable objection.
- **First, model the distribution of digit images**
    - [FWC - feature engineering!]
    - The top two layers (2000-top and 500-2nd-to-top) form a RBM 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
    - (Permutation Invariant : If we were to apply a fixed permutation to all the pixels in every training case, the results of our algorithm wouldn't change. (not true of a CNN))
    - Backprop net with one or two hidden layers (Platt; Hinton) - 1.6% error rate
    - Backprop with L2 constraints on incoming weights - 1.5% error rate
    - Support Vector Machines (Decoste & Schoelkopf, 2002) - 1.4% (this is one of the results that led SVMs to supplant NNs)
    - Generative model of joint (concatenation) density of images and labels (+ generative fine-tuning) - 1.25%
    - Generative model of unlabelled digits followed by gentle backpropagation (Generative pre-training, followed by discriminative fine-tuning; 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)
    - Architecture (top-to-bottom): 183 HMM-state labels (not pre-trained) <- 2000 logistic hidden units <- 6 more layers of pre-trained weights <- 2000 logistic hidden units <- 15 frames of 40 filterbank outputs & their temporal derivatives
    - 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

## Lecture 14c: What happens during discriminative fine-tuning?

- **Pre-training makes deep NNs more effective than shallower ones. W/out pre-training it's the other way around.**
- Learning Dynamics of Deep Nets
    - the next 4 slides describe work by Yoshua Bengio's group
    - pictures of the features before and after fine-tuning show no identifiable differences (but the results are better)
- Effect of Unsupervised Pre-training
    - Erhan et. al. AISTATS'2009
    - Red - with pretraining; Blue - w/out

    - With 1 layer - Red histogram of nets is almost completely below (less error) entire blue histogram (1.4% median vs. 1.8%)
    - Wtih 4 layers - Red is much lower, and blue has much higher stdev (1.3% median vs. 2.1%)
- Effect of network depth
    - W/out pretraining, 2 layers best, but quickly degrades
    - With pretraining, 4 layers best, but all more consistent and lower
- Trajectories of the learning in function space
    - (a 2-D visualization produced with t-SNE)
    - Erhan et. al AISTATS'2009
    - Each point is a model in *function space*
        - In order to compare 2 networks you have to compare the functions that they're implementing (because you can't compare neurons as 2 neurons might get swapped).
            - To do this you need a suite of test cases and look at the outputs produced on those test cases, and then concat outputs into 1 big long vector.
    - Color = epoch
    - Top: trajectories without pre-training. Each trajectory converges to a different local min.
    - Bottom: Trajectories with pre-training.
    - No overlap (in function space)!
        - The kinds of networks you find are just different w/ pretraining.
- **Why unsupervised pre-training makes sense**
    - [FWC - feature engineering!]
    - stuff-in-the-world/concept -> image -> label
        - 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?
    - stuff -> image & stuff -> label
        - high bandwidth from stuff in world to image, and low from stuff -> label
            - e.g. if I just say "cow" you don't know if the cow is upside down, alive/dead, facing towards/away; but if you see the image, you get much more information
        - 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.
    - It's more plausible that the reason an image has the name it has is because of the stuff in the world (the thing/concept the image is of or represents), not because of the pixels in the image.
    - [FWC - **this totally makes sense**]

## Lecture 14d: Modeling real-valued data with an RBM

- Make visible units linear w/ Gaussian noise, but this leads to problems w/ learning, which is solved by making the hidden units RELUs.
- 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. (e.g. very likely to be 0.69, but very unlikely to be 0.71 or 0.67)
            - [FWC - so why not learn differentials between a pixel and its surrounding pixels instead]
- **A standard type of real-valued visible unit**
- Model pixels as Gaussian (linear) variables. Alternating Gibbs sampling is still easy, though learning needs to be much slower.
    - E(v,h) = ∑_i[(v_i-b_i)^2/(2σ_i^2)] - ∑_j[b_j*h_j*] - ∑_*ij[v_i*h_j*w_ij / σ_i]
    - first term on RHS is "**parabolic containment function**" that keeps v_i near b_i
    - last term shifts that parabola away from b_i as determined by h

        - last term: energy-gradient produced by the total input to a visible unit
        - simple to get gradient of last term wrt v_i--a constant, which implies last term is a line
        - The effect of the hidden units is to push the mean to one side.
- 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** (see 0.69 example above)
        - It took a long time for us to figure out why it is so hard to learn the visible variances.
    - When sigma is much less than 1, the bottom-up effects (from v to h; w_ij/σ_i) are too big and the top-down effects (from h to v; σ_i*w_ij) are too small. [FWC - could a separate σ_i matrix be learned that gets applied oppositely in each phase?]
    - Solution: When sigma is small, we need many more hidden units than visible units.
        - This allows small weights to produce big top-down effects.
        - Unfortunately, we really need the number of hidden units to change with σ_i--cue Stepped sigmoid units
- 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 , ....
    - As σ_i gets smaller, the number of units that get turned on gets bigger resulting in more top-down effect to drive the visible units that have small stdevs.
- Fast approximations
    - It's quite expensive to use a big population of binary stochastic units with offset biases b/c for each one we need to compute the logistic function.
    - Contrastive divergence learning works well for the sum of stochastic logistic units with offset biases. The noise variance is sigmoid(y)
    - It also works for rectified linear units (ReLUs). These are much faster to compute than the sum of many logistic units with different biases/offsets.
    - <y> = ∑_{n=1 to ∞}[sigmoid(x+0.5−n) ≈ log(1+exp(x)) ≈ max(0, x+noise)
- 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 (e.g. to scale up the intensity of the image, just scale up the intensity of its repr)
        - R(a*x)* = *a*R(x) but R(a + b) ≠ R(a) + R(b)
    - It is like the translational equivariance exhibited by convolutional nets.
        - R(shift(x)) = shift(R(x))
        - The representation of a shifted image is just a shifted version of the representation of an unshifted image

## Lecture 14e: RBMs are Infinite Sigmoid Belief Nets

- Another view of why layer-by-layer learning works (Hinton, Osindero & Teh 2006)
    - There is an unexpected equivalence between RBMs 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 (with alternating hidden-visible layers and a repeating weights matrix)
        - 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 is equivalent to an RBM
- *Inference* (recall: "infer the states of the unobserved variables") in an infinite sigmoid belief net
    - The variables in h0 are conditionally independent given v0.
        - Inference is trivial. Just multiply v0 by W'
        - The model above h0 (i.e. starting at v1) implements a *complementary prior*: a prior (top-down) distribution over that exactly cancels out the (bottom-up) correlations in "explaining away."
        - **Multiplying v0 (FWC really v1, no? b/c priors work top-down) by W' gives the *product* of the likelihood term (FWC p(h0|v1)) and the prior term (FWC p(v1)), not just the likelihood term, and that's what you need to do to get the posterior.**

**It normally comes as a big surprise to people that when you multiply by W' it's the *product* of the posterior and the prior**
        - The complementary prior (everything above h0, computed top-down to v1) cancels the explaining away and makes inference very simple.
    - Inference in the directed net is exactly equivalent to letting an RBM settle to equilibrium starting at the data.
    - We can do inference for each layer and get an unbiased sample at each layer simply by multiplying v0 by W'. Then once we've computed the binary state of h0, we multiply that by W, put that through the logistic sigmoid and sample, and that will give us a binary state for v1, and so on, all the way up.
    - So just *generate* from this model is equivalent to running the alternating Markov chain of a RBM to equilibrium. Performing *inference* (recall: "infer the states of the unobserved variables") in this model is exactly the same process in the opposite direction. This is a very special SBN in which inference (bottom-up) is as easy as generation (top-down b/c data is at the bottom--i.e. we work top down to generate data). [FWC - here's another pointer explaining inference vs. generation in the form of a variational autoencoder which has an inference network (input-to-encoding) and a generation network (encoding-to-output)]
    - The learning rule for a sigmoid belief net is:
        - $\Delta w_{ij} \propto s_j(s_i - p_i)$ ... where j's are from hidden states and i's from visible
        - $s_i^1$ is an unbiased sample from $p_i^0$
    - The (inference) process that goes from h0 to v1 (up) is exactly the same as that (the generative process) which goes from h0 to v0 (down), which means that $s_i^1$ (**s**tate or **s**ample of visible unit i in second visible layer, v1) is an unbiased sample from $p_i^0$ (probability of visible unit i in first visible layer, v0)
    - With replicated weights this rule becomes:
        - $sj0(si0-si1) + si1(sj0-sj1) + sj1(si1-si2) + ... - sj\infty*si\infty$
        - rather than using p's (e.g. pi0) we use a sample with those probabilities (e.g. si1)
        - and so an unbiased estimate of the derivative can be gotten by plugging in si1 in that first term of the infinite sum learning rule
        - then for the second term: sj1 is an unbiased estimate of pj0
        - all of the terms except for the first and the last end up canceling out and you just end up with the Boltzmann machine learning rule
- Learning a deep directed network
    - First learn with all the weights tied. This is exactly equivalent to learning an RBM.
        - Think of the (bidirectional) 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 h0 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 its 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.**
    - Changing the weights makes the *inference* we're doing at the bottom layer incorrect, but gives us a better model.
- 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.
- Why is it OK to ignore the derivatives in higher layers?
    - When the weights are small, the Markov chain mixes fast. If they're 0 it mixes in 1 step.
        - 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!** A minima has been reached => no more learning can be done => derivatives of 0.
    - **As the weights grow we may need to run more iterations of CD (b/c it will take longer for the derivatives to fall to 0)**
        - This allows CD to continue to be a good approximation to maximum likelihood.
        - **If our purpose is to build a stack of RBMs for learning layers of features, it does not need to be a good approximation to maximum likelhood! In fact, it's probably better than maximum likelihood.**

## Lecture 14 Quiz

1. Why is a Deep Belief Network not a Boltzmann Machine?

- (CHECKED - INCORRECT: **A DBN has undirected edges in the top-layer RBM.**) - All edges in a DBN are directed.
- A DBN does not have hidden units.
- (CHECKED take 2) **Some edges in a DBN are directed.**
- A DBN is not a probabilistic model of the data.

2. Brian looked at the direction of arrows in a DBN and was surprised to find that the data is at the "output". "Where is the input ?!", he exclaimed, "How will I give input to this model and get all those cool features?" In this context, which of the following statements are true? Check all that apply.

- (UNCHECKED - INCORRECT the input is used prior to stacking the RBMs into a DBN but it can also be used to generate features (aka inference) by putting it in at the bottom where the output goes. INCORRECT) A DBN is a generative model of the data, which means that, its arrows define a way of generating data from a probability distribution, so **there is no "input"**. [FWC - because the input has already been learned by the stacked RBMs?]
- (CHECKED - INCORRECT put real data in at bottom, work up to RBM, iterate in top RBM layer, then generate back down. INCORRECT: This is not true. **A DBN can be used to generate features for any given data vector by doing inference.**) A DBN is a generative model of the data and cannot be used to generate features for any given input. It can only be used to get features for data that was generated by the model.
- (CHECKED - INCORRECT slide 33 of lec14.pdf. INCORRECT: Traversing arrows in the opposite direction is an *approximate* inference procedure.) In order to get features h given some data v, he must perform inference to find out P(h|v). There is an easy *exact* way of doing this, just traverse the arrows in the opposite direction.
- (UNCHECKED - INCORRECT) In order to get features h given some data v, he must perform inference to find out P(h|v). There is an easy *approximate* way of doing this, just traverse the arrows in the opposite direction.

3. In which of the following cases is pretraining likely to help the most (compared to training a neural net from random initialization) ?

- A dataset of binary pixel images which are to be classified based on parity, i.e., if the sum of pixels is even the image has label 0, otherwise it has label 1.
- A dataset of images is to be classified into 100 semantic classes. Fortunately, there are 100 million labelled training examples.
- A speech dataset with 10 billion labelled training examples.
- (CHECKED - b/c they're good at generating features - The small labelled set is likely to have enough information to learn a good word or n-gram model. The unlabelled set can be used to do pretraining for learning word embeddings.) - A dataset of movie reviews is to be classified.

There are only 1,000 labelled reviews but 1 million unlabelled ones can be extracted from crawling movie review web sites and discussion forums.

3. (different #3 on second try) Suppose you wanted to learn a neural net classifier. You have data and labels. All you care about is predicting the labels accurately for a test set. How can pretraining help in getting better accuracy, even though it does not use any information about the labels ?

- Pretraining will always learn features that will be useful for discrimination, no matter what the discriminative task is.
- The objective function used during pretraining is the same as the one used during fine-tuning. So pretraining provides more updates towards solving the same optimization problem.
- (CHECKED) There is an assumption that pretraining will learn features that will be useful for discrimination and it would be difficult to learn these features using just the labels. [FWC - what if all features had their labels "baked in" before pre-training s.t. what the pre-training was learning was relationships between features and labels? There would still be the same amount of information "bandwidth" but perhaps with label-knowledge as well]
- Pretraining will learn exactly the same features that a simple neural net would learn because after all, they are training on the same data set. But pretraining does not use the labels and hence it can prevent overfitting.

4. Why does pretraining help more when the network is deep?

- (CHECKED - CORRECT: Lower level layers can get very small gradients, especially if saturating hidden units are used (such as logistic or tanh units). Pretraining can initialize the weights in a proper region of weight space so that the features don't have to start learning from scratch.) During backpropagation in very deep nets, the lower level layers get very small gradients, making it hard to learn good low-level features. Since pretraining starts those low-level features off at a good point, there is a big win.
- (UNCHECKED) As nets get deeper, contrastive divergence objective used during pretraining gets closer to the classification objective.
- (CHECKED - CORRECT: More parameters means that the model can find ingenious ways of overfitting by learning features that don't generalize well. Pretraining can initialize the weights in a proper region of weight space so that the features learned are not too bad.) Deeper nets have more parameters than shallow ones and they overfit easily. Therefore, initializing them sensibly is important.
- (UNCHECKED) Backpropagation algorithm cannot give accurate gradients for very deep networks. So it is important to have good initializtions, especially, for the lower layers.

5. The energy function for binary RBMs goes by: E(v,h) = −∑_i[vi*bi]* − ∑*j[hj*aj] − ∑_ij[vi*Wij*hj]. When modeling real-valued data (i.e., when v is a real-valued vector not a binary one) we change it to: E(v,h) = ∑_i[(vi−bi)^2)/(2σi^2)] − ∑_j[hj*aj]* − ∑_*ij[vi*σi*Wij*hj] Why can't we still use the same old one ?

- (UNCHECKED) Probability distributions over real-valued data can only be modeled by having a conditional Gaussian distribution over them. So we have to use a quadratic term.
- (UNCHECKED - INCORRECT) If the model assigns an energy e1 to state v1,h, and e2 to state v2,h, then it would assign energy (e1+e2)/2 to state (v1+v2)/2,h. This does not make sense for the kind of distributions we usually want to model.
- (CHECKED - CORRECT it would just select huge positive vi's and hi's. Correct: If some bi<0, then if vi→−∞, then E→−∞. Similarly for bi>0 if vi→∞, then E→−∞. So the Boltzmann distribution based on this energy function would behave in weird ways.) If we continue to use the same one, then in general, there will be infinitely many v's and h's such that, E(v,h) will be infinitely small (close to −∞). The probability distribution resulting from such an energy function is not useful for modeling real data.
- (UNCHECKED) If we use the old one, the real-valued vectors would end up being constrained to be binary.

## Lecture 15a: From Principal Components Analysis to Autoencoders

- We can do PCA efficiently using standard methods, or we can do it inefficiently where both the hidden units and the output units are linear. The advantage of doing with a NN is that we can

generalize to deep NNs where the code is a non-linear function of the input and our reconstruction of the code is also a non-linear function of the code. This enables us to deal with **curved manifolds** in the input space.

- 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
- Using backpropagation to implement PCA *inefficiently* (although if there's a huge amount of data it may be more efficient)
  - Try to make the output be the same as the input in a network with a central bottleneck.
    - output vector <- code <- input vector
  - 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, but they may be a rotation/skewing of those axes.
    - Their weight vectors may not be orthogonal.
    - They will tend to have equal variances (unlike PCA).
- Using backpropagation to generalize PCA
  - With non-linear layers before and after the (bottleneck) code layer, it should be possible to efficiently represent data that lies on or near a non-linear manifold.
    - input -> encoder (logistic units) -> code (linear) -> decoder (logistic) -> output (matching input)
    - The encoder (between input and code) converts coordinates in the input space to coordinates on the manifold.
    - The decoder (between code and output) does the inverse mapping.
  - So we have a curious network in which we're using a supervised learning algorithm to do unsupervised learning.

## Lecture 15b: Deep Autoencoders

- People thought of these in the 80s, but they couldn't train them to be better than PCA.
- The Semantic Hashing (file:///home/fred/Documents/articles/autoencoders/semantic_hashing_salakhutdinov_hinton_09.pdf) article was the first time we could get much better results out of them than from PCA (actually it's a 2006 article by the same authors on MNIST digits; search this page for "2006"; see slide 8 of lec15.pdf)
- 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 (i.e. matrix mult)
  - 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)

    - MNIST 784 input pixels
    - stacked RBMs
    - 3 hidden encoding layers (1000, 500, 250)
    - 30 linear (code layer) units
    - 3 decoding layers (250 -> 500 -> 1000)
    - 784-pixel output layer
    - We train a stack of 4 RBM's (W1, W2, W3, W4) and then "unroll" them (transposes of Wi)
    - Then we fine-tune with gentle backprop.
        - So the weights for encoding diverged from those for decoding.
- A comparison of methods for compressing digit images to 30 real numbers (with pictures)
    - real data vs. 30-D deep auto vs. 30-D PCA
    - A linear method cannot do nearly as good a job at representing the data.
    - **FWC - i.e. don't use linear risk factors**

## [Lecture 15c: Deep autoencoders for document retrieval and visualization]

- "**10 components extracted with a DAE are worth as much as 50 components extracted with a linear method such as LSA**" (file:///home/fred/Documents/articles/autoencoders/reducing_dimensionality_w_NNs_hinton_salakhutdinov_2006.pdf and also see **Semantic Hashing**. Salakhutdinov, Hinton, 2009)
    - also showed that if you make the code vector very small (e.g. 2 components) you can use that for visualization in a 2D plot (which also works better than just extracting the first 2 principal components)
- 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”)--not much information
    - We could compare the word counts of the query document and millions of other documents but this is too slow (big vectors! 2000D in their case)
        - So we reduce each query vector to a much smaller vector that still contains most of the information about the content of the document.
- How to compress the count vector
    - 2000 word counts (bottom input vector) -> 500 neurons -> 250 neurons -> 10 numbers -> 250 neurons -> 500 neurons -> 2000 reconstructed counts (top output 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 (2 tricks)
    - First trick: 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.
    - Second trick: 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.
        - If we left them as probabilities, the input units would have very small activities (i.e. when multiplied by weights), and wouldn't provide much input to the first hidden layer.
        - So we have this funny property that **for the first RBM, the bottom-up weights are N times bigger than the top-down weights**.
- 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 (x) against the proportion that are in the same hand-labeled class as the query document (y).
        - Compare with LSA (a version of PCA).
- Retrieval performance on 400,000 Reuters business news stories
    - see slide 15 of lec15.pdf for plot
    - y = accuracy (proportional overlap wrt hand-labeled classes)
    - x = number of retrieved documents
- Slide 16 is a plot of all documents compressed to 2 numbers using *PCA* on log(1+count) (which supresses counds with very big numbers--and makes PCA work better), then colored for different (hand-labeled) categories.
    - There is *some* (i.e. minimal) separation of the classes.
- **Slide 17 is a plot of all documents compressed to 2 numbers using deep autoencoders (DAEs), then colored the same way.**
    - The different classes are distinct spokes in a wheel.
        - "We assume that the ones in the middle are documents that didn't have many words in them and therefore was hard to distinguish between the classes."
    - **The separation between the regions is amazing! Holy crap!**
    - "A plot like this could be very useful. For example if you saw one of those green (Accounts/Earnings) dots was for Enron, you probably wouldn't want to buy shares in a company nearby."
        - **FWC - similarly, if each of those points had a rating associated with it, you could select the nearest neighbors over a particular rating threshold** (think of the ratings as supervised labels)
    - **FWC - would be interesting to color the dots in one of these plots (wrt company characteristics) based on future returns** (This would be a good indicator of whether unsupervised pretraining can help with supervised prediction of some other variable. I.e. (1) perform unsupervised factorization, (2) plot, (3) color points based on supervised labels)

## Lecture 15d: Semantic hashing

- It's easy to get a binary description of an image such as black-white vs. color or inside-scene vs. outside-scene, but it's much more difficult to get 30 orthogonal binary bits
- Start by considering documents rather than images
    - Finding binary codes for documents
    - 2000D word counts -> 500 -> 250 -> 30 -> 250 -> 500 -> 2000 reconstructed softmax counts
    - Train an auto-encoder using 30 **logistic units for the code layer** (logistic units not sufficient b/c used in their middle ranges to convey as much information as possible--which the noise is added to prevent from happening)
    - 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
        - in forward pass use stochastic binary units for code layer
        - in backward pass pretend we used logistic units to get a smooth value for the gradient
    - To summarize:
        a. first train stacked RBMs
        b. then unroll by use transposes of RBM weight matrices for decoder
        c. then fine-tune backpropagate

        d. as we do that add additional Gaussian noise to the inputs to the code units (or easier way per Krizhevsky method above)
        - in order to be resistant to that noise, the code units need to be either firmly on or firmly off, and so the noise will encourage the learning to avoid the middle region of the logistic where it conveys a lot of information but is very sensitive to noise in its inputs
    - this allows us to convert the counts for a bag of words into a small number of binary values
    - so we can now compare a single query document against a long list of known documents (perhaps billions)
        - but there's a much faster thing we can do: we can treat the code as if it was a memory address (as in a hash)
        - so just go to a query memory address and find the documents to which it points
- Using a deep autoencoder as a hash-function for finding *approximate* matches
    - "supermarket search" - like what you'd do in an unfamiliar super market where you ask someone where the cans of anchovies are and they tell you where to find cans of tuna -- of course if you're unlucky they might be near the other pizza toppings, which is the downside of this kind of search
        - a supermarket though is only at most 2D - can't group by many different dimensions simultaneously (kosher, canned, out-of-date, vegetarian) -- but here we have a 30D supermarket
- Another view of semantic hashing
    - Fast retrieval methods typically work by **intersecting stored lists** that are associated with cues (e.g. a very rare word) extracted from the query.
        - So Google for example, will have a list of all the documents that contain some particular rare word, and so when you use that rare word in your query, they'll immediately have access to that list [FWC - seems similar to a forest or LSH)
        - FWC - couldn't you alternatively have a lot more code bits, say 10k, that are capable of representing all the possible intersections of such lists? or is the former method more flexible? or just easier to use the memory BUS?
    - Computers have special hardware that can intersect 32 very long lists in one instruction (the memory BUS)
        - 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 (with no search required at all)

## [Lecture 15e: Learning binary codes for image retrieval]

- **For document retrieval, places like Google have such good algorithms already, that techniques like Semantic Hashing may not be very useful** but different story for retrieving images (256 bits)
- 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! now we can identify objects with deep nets)
    - Maybe we should extract a real-valued vector that has information about the content of the image?
        - **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 (even faster)**
    - First, use semantic hashing with 28-bit binary codes to get a long "shortlist" of promising images (constant time jump to shortlist in memory).
        - Then use 256-bit binary codes to do a 1-by-1 serial search for good matches.
        - This only requires a few (4 for 256 bits) 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
  - Start w/ 32x32x3 (=3076) concatenated RGB vector for 32x32 pixel image but then expand (b/c logistic units have less capacity than real-valued) that to 8192 logistic units
  - Then halve the number of units in each layer until we get down to 256
  - The encoder has about 67,000,000 parameters--quite big.
    - It takes a few days on a Nvidia GTX 285 GPU to train on two million images.
  - **There is no theory to justify this architecture.**
    - We know we want to train a deep net, it makes sense to halve each layer, but interestingly a guess like this works quite well, and presumably there are others that will work better.
- Reconstructions of 32x32 color images from 256-bit codes
- Example retrievals (starting w/ a picture of Michael Jackson) using 256 bit codes
  - Starting with 256 bits, differing by only 61 bits is extremely unlikely to happen by chance, if they were random images.
  - Retrivals using euclidean distance on raw pixels picks up lots of non faces so obviously the autoencoder understands something about faces that's not picked up by Euclidean distance.
  - If you can't match the high frequency variation in an image, it's must better to match its average than other high frequency variation that's out of phase. **So when you've got a complicated image, Euclidean distance will typically find smooth images to match it** [FWC - kind of like linear vs. neural network based PCA]
- 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. [FWC - what about "semantic objects" in documents"
    - 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 (but obviously if it works for that then we can take those activity vectors and build an autoencoder on those)
    - It works really well!
    - Will it work with binary codes?
- Example images
  - Leftmost column is the search image.
  - Other columns are the images that have the most similar feature activities in the last hidden layer.
  - **These images would all have poor overlaps in pixel-space**
  - We'll see in lecture 16 that we can combine the content of the image with the caption to get an even better repr.

## Lecture 15f: Shallow autoencoders for pre-training

- Alternative pre-training methods for deep neural nets
- Pre-training was introduced w/ RBMs trained with CD, but after that people learned there are many other ways, and indeed if you initialize the weights correctly you may not need pre-training at all given enough labeled data.
- **RBMs can be viewed as shallow autoencoders, particularly if they're trained w/ contrastive divergence (CD)**.
- 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 (MLE), RBMs are not like autoencoders.
    - One way to see that is if you have a purely noisy pixel, an autoencoder would try to reconstruct whatever noisy value it had, while a RBM trained w/ MLE would

completely ignore that pixel and model it just using the bias for that input.
    - 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 [FWC - supervised learning]) if the shallow autoencoders are regularized by penalizing the squared weights, however, there's a different kind of AE that does work as well...
- ****** **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 rather than the hidden units).
        - They are still required to reconstruct these components so they must extract features that capture correlations between inputs.
        - The danger w/ a shallow AE is that, if you give it enough hidden units, it might just copy inputs to hidden units [FWC - too many degrees of freedom; over specification]
    - 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.
            - Can't compute the value of the *real* objective function for an RBM w/ CD (so we actually just use the squared reconstruction error, which isn't actually what's being minimized)
            - With a DAE we can print out the value of the thing we're trying to minimize--and that's very helpful.
        - It lacks the nice variational bound we get with RBMs, but this is only of theoretical interest (b/c only applies if RBM is trained w/ MLE)
- 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.
        - Obviously 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 for each region of input space.
        - RBMs behave similarly.
- Conclusions about pre-training (Hinton's current thinking)
    - 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!
        - FWC - This is assuming that large nets and small amounts of data (model/df size to data size ratio) are the only source of overfitting. There are other sources, of course, such as low R^2s, e.g. consider finance. Low R^2 only matters when predicting returns however, there is typically a lot of structure in other financial data, e.g. market cap vs. enterprise value.
    - Argument from Google: We don't need pre-training because we have so much labeled data now. Hinton: That's only because your nets are too small. Make them much bigger and you'll need pre-training again. FWC: Perhaps the Google people don't want pre-training b/c they don't think it's how the mind actually works. Hinton: The brain is clearly in the regime with huge amounts of data and very few labels. FWC: But maybe this is b/c we only thing about *positive* labels, i.e. what something is; we don't often consider negative labels, i.e. all the things something isn't.

## Lecture 15 Quiz

1. The objective function of an autoencoder is to reconstruct its input, i.e., it is trying to learn a function f, such that f(x)=x for all points x in the dataset. It makes sense to learn a mapping from x to some target t for solving a classification or regression problem, but why do we want to learn a mapping that takes x to x ? It seems like a silly thing to do!

- Due to the constraints in the network, x will never be mapped exactly to x but to something close-by, say x^. We can then use x^ as a better representation of x.
- We want to have a decoder network, so that it is easy to get a reconstruction of the data.
- CHECKED - While the objective is to reconstruct x, what we really care about is learning good representations in the hidden layers which come as a by-product of doing this.
- While the objective is to learn f such that f(x)=x, we might end up learning a mapping to a totally different space in which f(x) might be a much better representation of the input.

2. The process of autoencoding a vector seems to lose some information since the autoencoder cannot reconstruct the input exactly (as seen by the blurring of reconstructed images reconstructed from 256-bit codes). In other words, the intermediate representation appears to have less information than the input representation. In that case, why is this intermediate representation more useful than the input representation ?

- The intermediate representation actually has more information than the inputs.
- CHECKED - The intermediate representation may take much less memory to store and much less time to do comparisons with compared to the input. That makes matching queries to a database faster.
- The intermediate representation is more compressible than the input representation.
- The intermediate representation has more noise.

3. What are some of the ways of regularizing deep autoencoders?

- CHECKED (as in a Contractive AE) - Penalizing the squared gradient of the hidden unit activations with respect to the inputs.
- Using large minibatches for stochastic gradient descent.
- Using high learning rate and momentum.
- Using a squared error loss function for the reconstruction.

4. In all the autoencoders discussed in the lecture, the decoder network has the same number of layers and hidden units as the encoder network, but arranged in reverse order. Brian feels that this is not a strict requirement for building an autoencoder. He insists that we can build an autoencoder which has a very different decoder network than the encoder network. Which of the following statements is correct?

- Brian is correct, as long as the decoder network has at most as many parameters as the encoder network.
- Brian is mistaken. The decoder network must have the same architecture. Otherwise backpropagation will not work.
- CHECKED (INCORRECT) - Brian is correct, as long as the decoder network has the same number of parameters as the encoder network.
- CHECKED2 (CORRECT - An autoencoder is just a neural net trying to reconstruct its input. There is hardly any restriction on the kind of encoder or decoder to be used.) - Brian is correct. We can indeed have any decoder network, as long as it produces output of the same shape as the data, so that we can compare the output to the original data and tell the network where it's making mistakes.

5. Another way of extracting short codes for images is to hash them using standard hash functions. These functions are very fast to compute, require no training and transform inputs into fixed length representations. Why is it more useful to learn an autoencoder to do this ?

- Autoencoders have smooth objective functions whereas standard hash functions have no concept of an objective function.
- Autoencoders have several hidden units, unlike hash functions.

- CHECKED - Autoencoders can be used to do semantic hashing, where as standard hash functions do not respect semantics , i.e, two inputs that are close in meaning might be very far in the hashed space.
- For an autoencoder, it is possible to invert the mapping from the hashed value to the reconstruct the original input using the decoder, while this is not true for most hash functions.

6. **RBMs and single-hidden layer autoencoders can both be seen as different ways of extracting one layer of hidden variables from the inputs**. In what sense are they different?

- CHECKED (RBMs define a joint density P(v,h). Given v, we can compute P(h|v). Autoencoders define a function h=f(v)) - **RBMs define a probability distribution over the hidden variables conditioned on the visible units while autoencoders define a deterministic mapping from inputs to hidden variables.**
- CHECKED - **RBMs are undirected graphical models, but autoencoders are feed-forward neural nets.**
- CHECKED (The objective function for RBMs is log-likelihood. This is intractable to compute due to the partition function. Its gradients are hard to compute for similar reasons and computing them approximately requires CD or other MCMC methods. Autoencoders usually have tractable objectives such as squared loss or cross entropy which are easy to compute and differentiate.) - The objective function and its gradients are intractable to compute exactly for RBMs but can be computed efficiently exactly for autoencoders.
- RBMs work only with binary inputs but autoencoders work with all kinds of inputs.

7. Autoencoders seem like a very powerful and flexible way of learning hidden representations. You just need to get lots of data and ask the neural network to reconstruct it. Gradients and objective functions can be exactly computed. Any kind of data can be plugged in. What might be a limitation of these models?

- The inference process for finding states of hidden units given the input is intractable for autoencoders.
- CHECKED - **If the input data comes with a lot of noise, the autoencoder is being forced to reconstruct noisy input data. Being a deterministic mapping, it has to spend a lot of capacity in modelling the noise in order to reconstruct correctly. That capacity is not being used for anything semantically valuable, which is a waste.**
- The hidden representations are noisy.
- Autoencoders cannot work with discrete-valued inputs.
- CHECKED2 - There is no simple way to incorporate uncertainty in the hidden representation h=f(v). A probabilistic model *might* be able to express uncertainty better since it is being made to learn P(h|v).

## Final Exam

Final Exam

1. One regularization technique is to start with lots of connections in a neural network, and then remove those that are least useful to the task at hand (removing connections is the same as setting their weight to zero). Which of the following regularization techniques is best at removing connections that are least useful to the task that the network is trying to accomplish?

- Early stopping
- *CHECKED (correct) - L1 weight decay*
- (nope, same as L2) - Weight noise
- (nope, same as weight noise) - L2 weight decay

2. Why don't we usually train Restricted Boltzmann Machines by taking steps in the exact direction of the gradient of the objective function, like we do for other systems?

- (nope, there isn't an objective function, but there is an energy/loss function) - Because it's unsupervised learning (i.e. there are no targets), there is no objective function that we would like to optimize.
- *CHECKED (correct, this is why we aim for thermal equilibrium) - That gradient is intractable to compute exactly.*

- That would lead to severe overfitting, which is exactly what we're trying to avoid by using unsupervised learning.

3. When we want to train a Restricted Boltzmann Machine, we could try the following strategy. Each time we want to do a weight update based on some training cases, we turn each of those training cases into a full configuration by adding a sampled state of the hidden units (sampled from their distribution conditional on the state of the visible units as specified in the training case); and then we do our weight update in the direction that would most increase the goodness (i.e. decrease the energy) of those full configurations. This way, we expect to end up with a model where configurations that match the training data have high goodness (i.e. low energy). However, that's not what we do in practice. Why not?

- (this was my first incorrect answer) - The gradient of goodness for a configuration with respect to the model parameters is intractable to compute exactly.
- Correctly sampling the state of the hidden units, given the state of the visible units, is intractable.
- That would lead to severe overfitting, which is exactly what we're trying to avoid by using unsupervised learning.
- *CHECKED2 (CORRECT, this is true b/c the goodness/energy surface needs to be raised in places with low P data; only lowering it in places w/ high P data doesn't differentiate (aka contrastive divergence)) - High goodness (i.e. low energy) doesn't guarantee high probability.*

4. CD-1 and CD-10 both have their strong sides and their weak sides. Which is the main advantage of CD-10 over CD-1?

- The gradient estimate from CD-10 has more variance than the gradient estimate of CD-1.
- *CHECKED3 (CORRECT) - CD-10 gets its negative data (the configurations on which the negative part of the gradient estimate is based) from closer to the* **model distribution** *[FWC - the model distribution is that which is predicted by the model, the part of the energy surface we want to raise, as opposed to the data distribution!] than CD-1 does.*
- (my first incorrect answer) - The gradient estimate from CD-10 has less variance than the gradient estimate of CD-1.
- (my second incorrect answer) - CD-10 is less sensitive to small changes of the model parameters.
- The gradient estimate from CD-10 takes less time to compute than the gradient estimate of CD-1.

5. CD-1 and CD-10 both have their strong sides and their weak sides. Which are significant advantages of CD-1 over CD-10? Check all that apply.

- CD-1 gets its negative data (the configurations on which the negative part of the gradient estimate is based) from closer to the **model** distribution than CD-10 does.
- The gradient estimate from CD-1 has more variance than the gradient estimate of CD-10.
- *CHECKED4 [CORRECT FWC - maybe this should be checked if what they mean by "variance" is E_data - E_model] The gradient estimate from CD-1 has less variance than the gradient estimate of CD-10.*
- *CHECKED4 - The gradient estimate from CD-1 takes less time to compute than the gradient estimate from CD-10.*

6. With a lot of training data, is the perceptron learning procedure more likely or less likely to converge than with just a little training data? Clarification: We're not assuming that the data is always linearly separable.

- *CHECKED (correct, because it's less likely to be linearly separable) - Less likely.*
- More likely.

7. You just trained a neural network for a classification task, using some weight decay for regularization. After training it for 20 minutes, you find that on the validation data it performs much worse than on the training data: on the validation data, it classifies 90% of the data cases correctly, while on the training data it classifies 99% of the data cases correctly. Also, you made a plot of the performance on the training data and the performance on the validation data, and that plot shows that at the end of those 20 minutes, the performance on the training

data is improving while the performance on the validation data is getting worse. What would be a reasonable strategy to try next? Check all that apply.

- Redo the training with more hidden units.
- Redo the training with more training time.
- *CHECKED - Redo the training with more weight decay.*
- *CHECKED - Redo the training with fewer hidden units.*
- Redo the training with less weight decay.

8. *If the hidden units of a network are independent of each other, then it's easy to get a sample from the correct distribution* (FWC - recall factorial distribution?), which is a very important advantage. For which systems, and under which conditions, are the hidden units independent of each other? Check all that apply.

- [Answered this one incorrectly...twice...thrice...frice?]
- 1:ad 2:abd 3:d 4:bd 5:b 6:none .... todo ..... 7:bc 8:ab
    - In-class Questions
        - chapter 13 slide 24. What would be the gradient of the log probability of the data, for the recognition (red) weights?
            - A: 0. They are not involved in the data generation process.
        - What is the prior over the top hidden layer of an SBN like?
            - A: the logistic of the biases. The units are independent.
        - What is the prior over h of an RBM like?
            - A: to find out the probability of a particular hidden layer configuration, we have to consider all possible visible layer configurations. Therefore, this prior is not simple (as in SBNs).
        - **What is the posterior over h (given v) of an RBM? Simple or complex?**
            - *A: easy. clamp visible configuration. Every hidden configuration has its own energy (together with the visible configuration), which is easy to calculate. The hidden units are independent of each other, in this CONDITIONAL distribution.*
        - What is the posterior over h (given v) of an SBN with just one hidden layer? Simple or complex?
            - A: Explaining away makes the hidden units (conditionally) dependent, and this makes the distribution complex.
        - For getting an unbiased estimate of the gradient for an SBN, we need a sample from the posterior distribution over hidden units, given a visible units configuration from the training data. Do we need that, too, for getting an unbiased estimate of the gradient for a BM?
- (a) *SBN-NOTconditioned - CHECKED(1,2) (still think this is correct) - For a Sigmoid Belief Network where the only connections are from hidden units to visible units (i.e. no hidden-to-hidden or visible-to-visible connections), when we don't condition on anything, the hidden units are independent of each other.*
- (b) *RBM-conditioned CHECKED(2,4) (no b/c hiddens can never be independent when conditioned on visibles b/c of explaining away; perhaps should have been checked; nope, b/c an RBM is really an infinite SBN) - For a Restricted Boltzmann Machine, when we condition on the state of the visible units, the hidden units are independent of each other.*
- (c) SBN-conditioned (nope, b/c of explaining away) - For a Sigmoid Belief Network where the only connections are from hidden units to visible units (i.e. no hidden-to-hidden or visible-to-visible connections), when we condition on the state of the visible units, the hidden units are independent of each other.
- (d) RBM-NOTconditioned CHECKED(1,2,3,4) (still think this is correct) - For a Restricted Boltzmann Machine, when we don't condition on anything, the hidden units are independent of each other.

9. What is the purpose of momentum?

- *CHECKED (correct) - The primary purpose of momentum is to speed up the learning.*
- The primary purpose of momentum is to reduce the amount of overfitting.
- The primary purpose of momentum is to prevent oscillating gradient estimates from causing vanishing or exploding gradients.

10. Consider a Restricted Boltzmann Machine with 2 visible units v1,v2 and 1 hidden unit h. The visible units are connected to the hidden unit by weights w1,w2 and the hidden unit has a bias b. An illustration of this model is given below. The energy of this model is given by: E(v1,v2,h)=−w1v1h−w2v2h−bh. Recall that the joint probability P(v1,v2,h) is proportional to exp−E(v1,v2,h). Suppose that w1=1,w2=−1.5,b=−0.5. What is the conditional probability P(h=1|v1=0,v2=1)? Write down your answer with at least 3 digits after the decimal point.

- *A: 0.119*

11. Consider the following feed-forward neural network with one logistic hidden neuron and one linear output neuron. The input is given by x=1, the target is given by t=5, the input-to-hidden weight is given by w1=1.0986123, and the hidden-to-output weight is given by w2=4 (there are no bias parameters). What is the cost incurred by the network when we are using the squared error cost? Remember that the squared error cost is defined by Error=12(y−t)2. Write down your answer with at least 3 digits after the decimal point.

- *A: 2.000*

12. Consider the following feed-forward neural network with one logistic hidden neuron and one linear output neuron. The input is given by x=1, the target is given by t=5, the input-to-hidden weight is given by w1=1.0986123, and the hidden-to-output weight is given by w2=4 (there are no bias parameters). If we are using the squared error cost then what is ∂Error/∂w1, the derivative of the error with respect to w1? Remember that the squared error cost is defined by Error=0.5*(y−t)^2. Write down your answer with at least 3 digits after the decimal point.

- *A: -1.500*

13. Suppose that we have trained a semantic hashing network on a large collection of images. We then present to the network four images: two dogs (one brown with brown background, one light brown on green grass), a cat (same light brown with green background), and a car (blue car on green grass). The network produces four binary vectors: (a)[0,1,1,1,0,0,1] (b) [1,0,0,0,1,0,1] (c)[1,0,0,0,1,1,1] (d)[1,0,0,1,1,0,0] One may wonder which of these codes was produced from which of the images. Below, we've written four possible scenarios, and it's your job to select the most plausible one. Remember what the purpose of a semantic hashing network is, and use your intuition to solve this question. If you want to quantitatively compare binary vectors, use the number of different elements, i.e., the Manhattan distance. That is, if two binary vectors are [1,0,1] and [0,1,1] then their Manhattan distance is 2.

- (a)Dog 2(b)Dog 1(c)Car(d)Cat
- (a)Cat(b)Car(c)Dog 2(d)Dog 1
- *CHECKED (correct) - (a)Car(b)Dog 1(c)Dog 2(d)Cat*
- (a)Dog 1(b)Cat(c)Car(d)Dog 2

14. The following plots show training and testing error as a neural network is being trained (that is, error per epoch). Which of the following plots is an obvious example of overfitting occurring as the learning progresses?

- A: Third plot where test/training both drop (and converge) until 40 iters, then test begins to increase while training continues to drop.

15. Throughout this course, we used optimization routines such as gradient descent and conjugate gradients in order to learn the weights of a neural network. Two principal methods for optimization are online methods, where we update the parameters after looking at a single example, and full batch methods, where we update the parameters only after looking at all of the examples in the training set. Which of the following statements about online versus full batch methods are true? Check all that apply.

- *CHECKED - Mini-batch optimization is where we use several examples for each update. This interpolates between online and full batch optimization.*
- *CHECKED - Online methods scale much more gracefully to large datasets.*
- Full batch methods require us to compute the Hessian matrix (the matrix of second derivatives), whereas online methods approximate the Hessian, and refine this approximation as the optimization proceeds.

- Online methods require the use of momentum, otherwise they will diverge. In full batch methods momentum is optional.
- Full batch methods scale much more gracefully to large datasets.

16. You have seen the concept of **weight sharing** or **weight tying** appear throughout this course. For example, in dropout we combine an exponential number of neural networks with shared weights. In a convolutional neural network, each filter map involves using the same set of weights over different regions of the image. In the context of these models, which of the following statements about weight sharing is true?

- Weight sharing introduces noise into the gradients of a network. This makes it equivalent to using a Bayesian model, which will help prevent overfitting.
- Weight sharing implicitly causes models to prefer smaller weights. This means that it is equivalent to using weight decay and is therefore a form of regularization.
- Since ordinary convolutional neural networks and non-convolutional networks with dropout both use weight sharing, we can infer that convolutional networks will generalize well to new data because they will randomly drop out hidden units with probability 0.5.
- *CHECKED - Weight sharing reduces the number of parameters that need to be learned and can therefore be seen as a form of regularization.*

17. In which case is unsupervised pre-training most useful?

- CHECKED - ***** **There is a lot of unlabeled data but very little labeled data. [FWC - or very noisy labels, i.e. labeled data with little information content, it's not about how much labeled data there is, it's about how much information content it has]** *****
- There is a lot of labeled data but very little unlabeled data.
- The data is real-valued.
- The data is linearly separable.

18. Consider a neural network which operates on images and has one hidden layer and a single output unit. There are a number of possible ways to connect the inputs to the hidden units. In which case does the network have the smallest number of parameters? Assume that all other things (like the number of hidden units) are same.

- *CHECKED - The hidden layer is a convolutional layer, i.e. it has local connections and weight sharing.*
- The hidden layer is locally connected, i.e. it's connected to local regions of the input image.
- The hidden layer is fully connected to the inputs and there are skip connections from inputs to output unit.
- The hidden layer is fully connected to the inputs.

## Lecture 16a: Learning a Joint Model of Images and Captions

- Deep Restricted Boltzmann Machine, 1 for images, 1 for captions, then join at top

## Lecture 16b: Hierarchical Coordinate Frames

- 3 approaches to computer vision
  - deep CNN (best)
  - parts-based approach
  - using histograms of hand engineered features
- Why CNNs are doomed
  - When we pool, we lose exact location, which is fatal for higher level parts, like noses and mouths
  - Huge training sets with data at different transformations and scales -- clumsy
- Better way: hierarchy of coordinate frames
  - Use a group of neurons to repr the conjunction of the shape of a feature and its pose/orientation relative to the retina
  - Recognize the larger features by using the consistency of the poses/orientations of their parts
    - nose and mouth either make consistent or inconsistent prediction 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)
  - nose-node and mouth-node make face-node prediction which is activated if they both agree
  - in computer graphics we're going from poses of faces (larger things) to poses of their parts (components)
- A crucial property of the pose vectors
  - Allow spatial transformations to be modeled by linear ops
  - The invariant geometric properties of a shape are in the weights, not the activities
    - The activities are equivariant: As the pose of an object varies, the activities all vary
    - The percept of an object changes as the viewpoint changes
- Evidence that this is truly how the brain works (Irvin Rock)
  - The square and the diamond are different percepts that make different properties obvious
  - Percepts are different depending on what coordinate frame you impose

*** Lecture 16c: Bayesian optimization of neural network hyperparameters

- Use machine learning to replace the graduate student who typically fiddles around with lots of different settings of the hyperparameters to figure out what works
- When you're in a domain where you don't know much more than you expect similar inputs to have similar outputs, then Gaussian processes are ideal for finding good sets of hyperparameters
- Let machine learning figure out the hyper-parameters (Snoek, Larochelle, Adams, NIPS 2012)
  - One of the commonest reasons for not using NNs is that it requires a lot of skill to set hyperparameters
    - number of layers
    - # units / layer
    - type of unit
    - weight penalty
    - learning rate
    - momentum
    - dropout
  - One approach: naive grid search: Make list of alternative values for each hyperparameter and try all possible combos
    - This is slow. Can we do better?
  - Sampling random combinations
    - This is much better if some hyperparameters have no effect
    - It's a big waste to exactly repeat the settings of the other hyperparameters
- 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 HP space that might give better results
    - We need to predict how well a certain region will do, but we also need to model the uncertainty
  - We assume the amount of computation needed to explore one region of HP space is huge
    - Much more than the work involved in building a model that predicts the result from knowing previous results with different settings
- Gaussian Process models
  - Assume similar inputs give similar outputs
    - Very weak, but sensible prior
  - For each input dimension, they learn the approprate scale for measuring similarity
    - Is 200 similar to 300?
    - Look to see if they give similar results for the data so far

    - GP models do more than predict the mean, they predict variance also, they predict a Gaussian distribution
        - For test cases that are close to several consistent training cases the predictions are quite sharp
- A sensible way to decide what to try
    - Keep track of the best setting so far
    - Pick a setting of the HPs st that the expected improvement in our best setting is big
        - don't worry about the downside (hedge funds! 2% performance fee if we make money, but 0% downside if we don't)
        - big expected improvement means 2 things
            a. a parameter is far from its expected best value
            b. the parameter has high variance (i.e. changing it has a high P of helping)
- How well does Bayesian optimization work?
    - If you have a lot of resources to run lots of experiments, it's better than a human.
    - This is not the kind of thing we are good at.
    - Might fail to notice that all good results have small LRs while bads have high LRs
    - They also don't cheat: no bias for a model a human likes (e.g. b/c he thought of it) vs. one he doesn't (b/c he didn't think of it)