Notes on neural networks

Michael Nielsen[1,2]

July 11, 2013

[1]Email: mn@michaelnielsen.org
[2]Web: http://michaelnielsen.org/ddi

Chapter 1

Introduction

Working notes, by Michael Nielsen: These are rough working notes, written as part of my study of neural networks. Note that they really are *rough*, and I've made no attempt to clean them up, nor do I plan to. They contain misunderstandings, misinterpretations, omissions, and outright errors. As such, I don't advise others to read the notes, and certainly not to rely on them!

Core questions: There is a practical, narrow question: what are the most significant results about deep learning and neural networks? And then there is the broader question: how to build an artificial intelligence? My reading will address both questions.

Chapter 2

Papers

2.1 Hopfield (1982)

What I like about this paper is the condensed matter physicist's point of view. Hopfield asks "whether the ability of large collections of neurons to perform computational tasks may in part be a spontaneous collective consequence of having a large number of interacting simple neurons". He goes on to give an explanation of how a type of memory can be constructed in pretty much this way. It's an inspiring point of view.

2.2 Bourland and Kamp (1988)

link

Suggests removing nonlinearity in output. Motivation: since we're trying to recover the original input, claims that it's obviously not a good idea to have the nonlinearity. I don't see that this is true: if the inputs are normalized to be between 0 and 1 then there shouldn't be any problem.

With this constraint, the problem then is to find w, b and w', b' minimizing:

$$\sum_x \|w'\sigma(wx + b) + b' - x\|^2, \tag{2.1}$$

where the sum is over all input vectors x. Let X be the matrix whose columns are the training vectors. Abusing notation, let b and b' be matrices whose columns are b and b', respectively. Then matrix whose columns are

the outputs is given by $Y = w'\sigma(wX+b)+b'$, where we apply σ elementwise to the matrix $wX + b$. The quadratic loss function can then be written:

$$\|w'\sigma(wX + b) + b' - X\|^2, \tag{2.2}$$

where $\|\cdot\|$ is here the usual Frobenius matrix norm.

2.3 Blum and Rivest (1989)

Show that it's NP-complete to train a three node neural network. Apparently built on earlier work by Judd, who showed this for a general neural network; indeed, Judd showed that even approximating a function is NP-complete. Blum and Rivest use a very particular architecture: n inputs, a 2-neuron hidden layer, and a single output neuron. They use a perceptron model, although I doubt that is essential. The idea is simply to take a (supervised) training problem, and to ask whether there exist weights so that the output from the network are consistent with the training data. They show that this problem is NP-complete. They contrast this with a single-layer perceptron, which can be trained in polynomial time, using linear programming. They comment that their technique does not apply to sigmoidal neurons, but that Judd's does.

2.4 Williams and Zipser (1989)

(link) A gradient-based learning method for recurrent neural networks.

Claims that feedforward networks don't have the "ability to store information for later use". It'd be nice to understand what that means. Obviously there's a trivial sense in which feedforward networks can store information based on training data.

Claims that backprop requires lots of memory when used with large amounts of training data. I don't believe this, except in the trivial sense that it may take a lot of memory to store all the training data. Otherwise, we can compute gradients training-instance-by-training-instance, and sum the results, which is not especially memory intensive. (Of course, one may have a huge network which requires a lot of memory to story. But that's a separate issue.)

Their model of recurrent neural networks is interesting. Basically, we have a set of neurons, each with an output. And we have a set of inputs to the network. There is a weight between every pair of neurons, and from each input to each neuron. To compute a neuron's output at time $t+1$ we compute the weighted sum of the inputs and the outputs at time t, and apply the appropriate nonlinear function (sigmoid, or whatever). Note that in order for this description to make sense we must specify the behaviour of the external inputs over time. We can incorporate a bias by having an external input which is always 1.

So a recurrent neural network is just like a feedforward network, with a weight constraint: the weights in each layer are the same, over and over again. Also, the inputs must be input to every layer in the network.

Williams and Zipser take as their supervised training task the goal of getting neuron outputs to match certain desired training values at certain times. For instance, you could define a two-neuron network that will *eventually* produce the XOR of the inputs.

They define the total error to be the sum over squares of the errors in individual neuron outputs. And we can then do ordinary gradient descent with that error function. They derive a simple dynamical system to describe how to improve the weights in the network using gradient descent.

The above algorithm assumes that the weights in the network remain constant for all time. Williams and Zipser then modify the learning algorithm, allowing it to change the weights at each step. The idea is simply to compute the total error at any *given* time, and then to use gradient descent with that error function to update the weights. (Similar to online learning in feedforward networks.)

Williams and Zipser describe a method of *teacher-forcing*, modifying the neural network by replacing the output of certain neurons by the *desired* output, for training purposes in later steps.

Unfortunately, it is still unclear to me *why* one would wish to use recurrent neural networks. Williams and Zipser describe a number of examples, but they don't seem compelling.

The algorithm in which the weights can change seems non-physiological — it verges on being an unmotivated statistical model. (I doubt that the weights in the brain swing around wildly, but I'll bet that the weights found by this algorithm can swing around wildly.) The algorithm in which the weights are fixed seems more biological.

Note that Williams and Zipser *do not* offer any analysis of running time

for their algorithms, or an understanding of when it is likely to work well, and when it is not. It's very much in the empirical let's-see-how-this-works style adopted through much of the neural networks literature.

Summing up: the recurrent neural network works by, at each step, computing the sigmoid function of the weighted sum of the inputs and the previous step's outputs. Training means specifying a set of desired outputs at particular times, and adapting the weights at each time-step. Training works by specifying an error function at any given time step, computing the gradient, and updating the weights appropriately.

2.5 Baldi and Hornik (1989)

(link) This characterizes linear autoencoders. We have a three-layer network, and the output is related to the input by $x \to ABx$, where B describes the first layer of weights, and A the second layer. The goal is to find weight matrices A and B to minimize:

$$\sum_x \|x - ABx\|^2. \tag{2.3}$$

The challenge is that the hidden layer has a *smaller* number h of neurons than the input layer (which is, of course, of the same size as the output layer)[1]. Let me try an attack on this without reading the paper. That sum above is just:

$$\mathrm{tr}((I - AB)^2 \Sigma), \tag{2.4}$$

where $\Sigma \equiv \sum_x xx^T$. To minimize this what we want to do is obvious (and easily proven): we'll choose A and B so that AB is a h-dimensional projector onto the span of the eignenvectors of Σ with the h largest eigenvalues. Let $P(\Sigma, h)$ denote such a projector, so:

$$AB = P(\Sigma, h). \tag{2.5}$$

We can easily characterize such A and B. A should take the space $P(\Sigma, h)$ into the space spanned by the outputs from the hidden units, and B should

[1] It's not quite clear to me what h should parameterize. I'll use it to parameterize the number of dimensions in the vector space representing outputs from the hidden units. It seems likely that it'd be better to write 2^h, but I'll ignore that.

then undo that transformation. There is an orthogonal freedom inbetween time, and a possible freedom in $P(\Sigma, h)$. This completely characterizes A and B.

Summing up, in a linear neural network, *a linear autoencoder is just doing principal components analysis.* So *a non-linear autoencoder can be thought of as a non-linear generalization of PCA.* That's a useful fact to remember. Examination of the remainder of the paper suggests that these are the key facts.

2.6 Olshausen and Field (1996)

Presents a method for finding low-complexity representations of natural images, in terms of atomic images — which they call "sparse codes" — which are localized, oriented, and scale-sensitive. These are found using an unsupervised learning algorithm with a bias toward good quality, low-complexity representations. The codes seem to be quite similar to the receptive fields found in the human visual system.

The *receptive field* for a cell in the retina is the volume of space (roughly, a cone) which can stimulate that cell to fire. Nearby cells can have overlapping (or nearby) receptive fields. Other cells in the visual cortex also have receptive fields, but they may be more complex, since the light has already been filtered through one or more levels of processing.

The paper claims that the receptive fields in the primary visual cortex are: (a) spatially localized; (b) oriented; and (c) can distinguish structure at different scales.

There is then a question: so what are those receptive fields? In a way, we can view this as being the question: to what type of images do different cells in our primary visual cortex respond? Answering that question seems like a good start for understanding any higher-level image processing. It's the question: what are the atoms of image processing? Or perhaps a better way is to think of them as the molecules of image processing, since they're one level up from the pixel level.

They develop an unsupervised learning algorithm which, trained on natural data, can find receptive fields that are spatially localized, oriented, and can distinguish structure at different scales.

Olshausen and Field want to decompose an image as:

$$I(x, y) = \sum_j a_j \phi_j(x, y). \tag{2.6}$$

The idea is that the ϕ_j form a (possibly overcomplete) basis for the space of images. They want to choose the ϕ_j which "results in the coefficient values being as statistically independent as possible over an ensemble of natural images". In some sense, the different a_j would be "telling us different things" about the image. They also want the coefficient values to be sparse, favouring simple representations over more complex.

O & F try to search for a suitable set of ϕ_js by introducing an error function:

$$E = -[\text{preserve information}] - \lambda[\text{sparseness of } a_j]. \tag{2.7}$$

This error is *for a single image*. The first term is just the l_2 error, i.e., (minus) the quadratic distance between the image and its representation. The sparseness term is just a nonlinear function of the a_j coefficients, quantifying how sparse they are.

The idea is to do online learning with this error function, presenting it with natural images, and gradually minimizing the error. (I see later in the article that it was actually batch learning using conjugate gradient descent. It appears that some kind of average error is being computed.) The result will be an overcomplete basis set that favours sparse decompositions of images.

The "sparsification" idea is a very interesting one. Basically, it's a way of trying to force a kind of Occam's razor into the system. It's a bit like autoencoders, forcing a simple explanation of complex data.

O & F note that wavelets have been used to find sparse codes previously.

2.7 LeCun (1998)

link

Reviews the classic two-part architecture: a feature extraction module, followed by a trainable classifier module. Points out that the real goal is to shunt as much as possible out of the feature extraction module and into the classifier module, since the first requires hand-engineering, while the second is (much more) automated.

Makes the remarkable claim that the difference in error between test and training set scales as $(h/N)^\alpha$, where h is a measure of how complex a classifier we're using, N is the number of training examples, and $0.5 < \alpha < 1$. In other words, the error grows as the complexity of the machine grows. And it shrinks as the number of training samples grows. I wonder why this is the case? Could we come up with a model that more or less proves that this is the case? Maybe a renormalization argument?

"The fact that local minima do not seem to be a problem for multi-layer neural networks is somewhat of a theoretical mystery": This is strange. Maybe it's the case that it's very hard to fall down into such local minima in high dimensions? I've personally had problems with very simple training data, but as soon as the training data and network become at all complex, those problems seem to vanish. This presumably means that "most" local minima are pretty darn good.

The *segmentation* problem: the problem of cutting up a string of characters. Notes that a nice heuristic is to try lots and lots of different cuts, and for each possible cut to score the cut by using the individual character classifier: if that classifier seems to be working well, then chances are that you have a good cut.

The authors note that existing systems are based on hand-crafted feature extractors, but that they will not use hand-crafted features.

MNIST: constructured by combining NIST Special database 3 (SD-3) and Special Database 1 (SD-1). Apparently, NIST designated SD-3 as a training set, and SD-1 as a test set. But the two are actually very different from on enaother. SD-3 is a clean data set, taken from Census Bureau employees, while SD-1 is not so good, being taken from high-school students. They describe some details of how MNIST was constructed. I'll review a few particularly striking facts. First, each character is size normalized, while preserving aspect ratio, and centred. There was also anti-aliasing going on. So this can all be regarded as pre-processing of features. The database was prepared in three forms. One was the form I know it. A second was a deslanted form. The third reduced the image resolution.

Deslanting: Idea was to compute moments of inertia, and then to recenter things (vertically), while downsampling to 20 by 20. As we'll see below this significantly improves performance.

Convolutional networks: They use local receptive fields, shared weights, and spatial sub-sampling. "With local receptive fields, neurons can extract elementary visual features such as oriented edges, end-points, corners (or

similar features in other signals such as speech spectrograms). These features are then combined by the subsequent layers in order to detect higher-order features." "... elementary feature detectors that are useful on one part of the image are likely to be useful across the entire image. This knowledge can be applied by forcing a set of units, whose receptive fields are located at different places on the image, to have identical weight vectors."

"Units in a layer are organized in planes within which all the units share the same set of weights". So the basic idea is to convolve the original inputs in some small window of the inputs. We call this a "feature map". I think Hinton later calls it a kernel(?) We will typically have several different feature maps. So what we have is a convolution stage. FOr example, we might have a 5 by 5 feature map. This is applied to a 5 by 5 receptive field in the input, i.e., a 5 by 5 area in the input. Each unit has 25 inputs, and so 25 weights and a bias. "all the units in a feature map share the same set of 25 weights and the same bias so they detect the same feature at all possible locations on the input." "The other feature maps in the layer use different sets of weights and biases, thereby extracting different types of local features." In LeNet-5 there are 6 feature maps. Note that a squashing function and bias apparently are used — this wasn't apparent earlier, where the focus is on the convolution. Note that the feature map output will respect translations of the original image.

Sub-sampling: The intuition is that exact location information is not necessary. "Not only is the precise position of each of those features [identified by the feature maps] irrelevant for identifying the pattern, it is potentially harmful because the positions are likely to vary for different instances of the character." "A simple way to reduce the precision with which the position of distinctive features are encoded in a feature map is to reduce the spatial resolution of the feature map. This can be achieved with a so-called sub-sampling layers [*sic*] which performs a local averaging and a sub-sampling, reducing the resolution of the feature map, and reducing the sensititivity of the output to shifts and distortions." In LeNet-5 they use a sub-sampling layers, which perform a kind of local averaging and sub-sampling. Basically, they use six 2 by 2 features maps, one for each of the previous six feature maps. "Each unit computes the *average* of its four inputs, multiplies it by a trainable coefficient, adds a trainable bias, and passes the result through a sigmoid function". It's notable here that we don't have trainable weights in the ordinary fashion. It's also notable that things aren't overlapping in this case, unlike the local receptive fields. Possibilities for this layer: blurring,

local max, local min. (Depends on parameter values).

Architecture: "Successive layers of convolutions and sub-sampling are typically alternated..." Traces the origins of the idea to Hubel and Wiesel and to Fukushima. It sounds as though the main new thing here is to try it out with backprop. The paper also describes some previous applications of convolutional neural networks to image and speech recognition.

"Since all the weights are learned with back-propagation, convolutional networks can be seen as synthesizing their own feature extractor." Big advantage of reducing the number of parameters: it reduces overfitting.

LeNet-5: 7 layers, not counting the input. 32 by 32 inputs. Note that the characters are themselves 20 by 20 pixels centered in a 28 by 28 field.

Layer C3 (third layer, convolutional): 16 feature maps. Each unit in each feature map is connected to several 5 by 5 neighbourhoods are identical locations in a subset of S2's feature maps. "WHy not connect every S2 feature map to every C3 feature map?" (1) Reduce the number of connections; (2) Forces a break in symmetry in the network. My guess is that it would otherwise work, but might be slower. "Different feature maps are forced to extract different (hopefully complementary) features because they get different sets of input."

Layer C5: 120 feature maps. Each unit is connected to a 5 by 5 neighbourhood on all 16 of S4's feature maps. They state that this amounts to a full connection between S4 and C5 — this is true because each feature unit is just a single unit.

They use a scaled hyperbolic tangent as the squashing function. "As seen before, the squashing function used in our Convolutional Networks is $f(a) = A\tanh(Sa)$. Symmetric functions are believed to yield faster convergence [i.e., learn at a faster rate], although the learning can become extremely slow if the weights are too small. The cause of this problem is that in weight space the origin is a fixed point of the learning dynamics, and, although it is a saddle point, it is attractive in almost all directions". It seems likely to me that we will have a similar problem with the usual sigmoid function. They chose their parameters to ensure $f(\pm 1) = \pm 1$, i.e., for convenience. "This particular choice of parameters is merely a convenience, and does not affect the result."

They initialize weights with the inverse of the fan-in, omitting the square root that I am accustomed to use. "The standard deviation of the weighted sum scales like the square root of the number of inputs when the inputs are independent, and it scales linearly with the number of inputs if the inputs

are highly correlated. We choose to assume the second hypothesis since some units receive highly correlated signals." The second clause in the first sentence is simply false, since the weights are set independently of the inputs. It's interesting that their method apparently works okay anyway, i.e., it must be quite insensitive to this detail.

Final layer in the network: Euclidean Radial Basis functions (RBF), one for each class (i.e., 10 in total), with 84 inputs. The output is the squared Euclidean distance between the inputs and the input weights. In other words, the RBF measures how close the input is to the weights. Fascinatingly, the initial values for these were set by hand, based on very simple versions of ASCII characters.

"[O]utput units... must be off most of the time. This is quite difficult to achieve with sigmoid units." Not sure why.

Learning schedule: $\eta = 0.0005$ for the first two epochs, 0.0002 for the next three, 0.0001 for the next three, 0.00005 for the next four, and 0.00001 for the remaining epochs (up to 20, so it was eight).

Distortions: "When distorted data was used for training, the test error rate dropped to 0.8 percent (from 0.95 percent without deformation)." It'd be nice to have a nice little library of transformations.

Linear classifier: 12 percent error rate. When deslanted, gets 8.4 percent error rate. "Various combinations of sigmoid units, linear units, gradient descent learning, and learning by directly solving linear systems gave similar results". "A simple improvement of the basic linear classifier was tested. The idea is to train each unit of a single-layer network to separate each class from each other class. In other words, there are $\binom{10}{2} = 45$ units. There is still a need to have a final decision procedure, and they simply chose the class which beat the largest number of other classes. "The error rate on the regular test set was 7.6%".

Baseline nearest neighbor classifier: Using Euclidean distance between input images. "On the regular test set the error rate was 5.0%. On the deslanted data, the error rate was 2.5%, with $k = 3$."

PCA: Computes the projection of the input pattern on the 40 principal components. "The 40-dimensional feature vector was used as the input of a second degree polynomial classifer." "The error on the regular test set was 3.3%."

Radial basis functions: Error rate of 3.6%.

One-hidden layer neural network: Error was 4.7% for a network with 300 hidden units. Interesting: this is worse than my results, even when I'm using

mean-square error (I get some improvement from using cross-entropy). I don't know why. My initialization is somewhat different. Otherwise, I can't think of any reason. They get a reduction to 4.5% for a network with 1000 hidden units(!) They did even bettter with distortions: 3.6% and 3.8%, with 300 and 1000 hidden units, respectively. When deslanted images were used, the test error dropped to 1.6%, with 300 hidden units. Raises the question of why we don't get terrible overfitting, just on parameter counting grounds.

Two-hidden layer neural network: "The test error rate of a 784-300-100-10 network was 3.05%, a much better result than the one-hidden layer network [4.7%], obtained using marginally more weights and connections." This doesn't accord with my experience using basic backprop. Rather, it's like their results now match up with mine for both a single and two-hidden layer. (Admittedly, I do get an improvement — albeit more modest — if pretraining is used). However, I'm using both the cross-entropy and different weight initialization. So identical results wouldn't be expected. Increasing the network size to 784-1000-150-10 improved things only a tiny bit, to 2.95%. Training with distorted patterns improved things to 2.5% and 2.45%, respectively.

LeNet-1: A small convolutional net. It got 1.7% test error rate. "The fact that a network with such a small number [2,600] of parameters can attain such a good error rate is an indication that the architecture is appropriate for the task."

Boosting: This is a technique which sounds like an idea I've been wondering about: concentrating more on training data which the network is misclassifying.

Tangent distance classifier: This is an interesting idea. The idea is to consider the tangent plane near a digit image, where we're considering a (low-dimensional) submanifold generated by distortions and translations of the images. "An excellent measure of 'closeness' for character images is the distance between their tangent planes, where the set of distortions used to generate the planes includes translations, scaling, skewing, squeezing, rotation, and line thickness variations". They use this measure of distance to run a nearest-neighbor method classifier. They get an error rate of 1.1%, which is (obviously) excellent.

Support vector machines: Depending on technique, results obtained varied between 1.4% and 0.8%.

They report on the number of operations required to do a classification, and the convolutional networks do quite well. Much better than the SVMs, interestingly enough, perhaps because the SVMs are fitting high-order poly-

nomials, and thus have a very large number of terms.

"When plenty of data is available, many methods can attain respectable accuracy. The neural-net methods run much faster and require much less space than memory-based techniques. The neural nets' advantage will become more striking as training databases continue to increase in size."

Invariance and noise resistance: "Convolutional networks are particularly well suited for recognizing or rejecting shapes with widely varying size, position, and orientation, such as the ones typically produced by heuristic segmenters in real-world string recognition systems. In an experiment like the one described above, the importance of noise resistance and distortion invariance is not obvious. The situation in most real applications is quite different. Characters must generally be segmented our of their context prior to recognition. Segmentation algorithms... often leave extraneous marks in character images... or sometimes cut characters too much and produce incomplete characters. Those images cannot be reliably size-normalized and centered. Normalizing incomplete characters can be very dangerous. For example, an enlarged stray mark can look like a genuine 1."

Conclusions: "Convolutional Neural Networks have been show to eliminate the need for hand-crafter feature extractors. Graph Transformer Networks have been shown to reduce the need for hand-crafted heuristics, manual labeling, and manual parameter tuning in document recognition systems." "It was shown that all the steps of a document analysis system can be formulated as graph transformers through which gradients can be back-propagated." "It is worth pointing out that data generating models... and the Maximum Likelihood Principle were not called upon to justify most of the architectures and training criteria described in this paper."

2.8 Ferret rewiring (Nature, 2000)

The primary visual cortex has what are called orientation modules. These are groups of cells that share a preferred "stimulus orientation". It's not clear to me what a stimulus orientation is, exactly — do they mean the direction the stimulus comes from. I'll get back to that. Anyway, there is apparently an orientation map. Well, when they rewire the ferrets' brains, apparently there are visually responsive cells in the auditory cortex that start to develop an orientation map! It's similar to the one in the visual cortex, although apparently less orderly.

They use a nice piece of terminology: sensory pathways have an *instructive* role in the development of cortical networks. The visual cortex apparently has a couple of different kinds of structure: ocular dominance columns, and orientation columns. Actually, looking at Wikipedia, there's quite a bit more structure in there than that. Apparently, orientation columns were discovered simply by stimulating a cat with visual stimuli from different directions, and noticing where in the visual cortex excitement occurred. They're apparently little slabs of cells that respond to visual stimuli from a particular direction. Perhaps unsurprisingly, these columns are arranged into little pinwheels — it's natural enough that they would reflect external geometry.

They wanted to investigate "whether afferent [i.e., sensory] activity or intrinsic features of the cortical target regulate the development of orientation columns." "... within limits, input activity [from eyes to auditory cortex] has a significant instructive role in establishing the cortical circuits that underlie orientation selectivity and the orientation map".

They identify two separate things — the degree of "tuning" in the cortex, as well as the orientation map. Apparently, these two things are found to be more or less independent. What's "orientation tuning" mean? Maybe it's a way of calibrating the respective meaning of activation of different orientation columns? "... afferent activity is required for at least the maintenance of orientation selectivity in V1 neurons". In other words, you destory the orientation structure if you don't get sensory input. This is a complementary result.

2.9 Tenenbaum, de Silva and Langford (2000)

(link) They mention a technique called multidimensional scaling (MDS), which I hadn't heard of. The idea seems to be that we have a lot of items, and we know some "dissimilarities" between items. The goal is to find a metric space embedding of those items so that the distances are roughly equal to the dissimilarities.

A sample problem: we have a 4096-dimensional space, corresponding to 64 by 64 pixel images. A (nonlinear) subspace of this corresponds to images we'd recognize as faces. How can we characterize this subspace?

This is just one possible mathematical formalization of the problem. In practice, things are more complex. Our classification will be fuzzy. We'll have all kinds of extra contextual information: maybe we've got an external

hint; maybe we can see a nose; maybe the colour is wrong, but we see enough to suspect it's false colour. All these kinds of things are clearly important in how we actually see. In other words, we don't just have an algorithm for face detection. We have a million related algorithms, and they all affect how well face detection works. In some sense you don't solve one problem perfectly. You solve a network of problems imperfectly — and then use those results to improve your performance on the original problem. It's a kind of *learning network*. In a sense this is what a deep neural network does: it builds up gradually more complicated features.

The algorithm they describe is very simple. Very roughly (this certainly contains mistakes): the idea seems to be to take all your data points and to compute distances between them. We assume that when the distances are small, the points are neighbours. Construct a graph in which neighbouring points are connected. Then geodesic distance is found (approximated) by finding the shortest distance in the graph. We then embed the graph in a space of the chosen dimensionality. Nice! Simple, probably pretty easy to implement, and I expect it lets us find a lot of structure.

It's worth thinking about what the input and output are. The input to Iso-map is just a data set — maybe it's a set of images of a face, maybe it's a set of words, whatever. This data lives in a very high-dimensional space. What we do is we find an embedding in a much lower dimensional space — say, 2-dimensional. In other words, we're constructing new features, based on the original features.

There are 10^6 optic nerves and $30,000$ auditory nerves: I'm not quite sure what to make of this. Presumably it means that we process something like 30 times as much optical information as auditory. I wonder how pixellated the information is?

What happens when we augment the features, with PCA? Let's suppose we start off with 3 features, x, y, z. Then we add x^2 and y^2 as new features. Certain subsets of the original space that weren't linearly approximable *will be* in the new feature space. This seems like a potentially powerful technique. What can it be used to do? What are its limits?

2.10 Simard (2003)

"Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis"

"The most important practice is getting a training set as large as possible: we expand the training set by adding a new form of distorted data". They claim it's better even than being convolutional. "The optimal performance on MNIST was achieved using two essential practices. First, we created a new, general set of elastic distortions that vastly expanded the size of the training set..."

"We avoided using momentum, weight decay, structure-dependent learning rates, extra padding around the inputs, and averaging instead of subsampling. (We were motivated to avoid these complications by trying them on various architecture/distortions combinations and on a train/validation split of the data and finding they did not help.)"

They have lots of useful details about how they came up with their convolutional architecture. It's very similar to LeCun (1998), of course, but they have more detail on *how* they chose the various parameters. Interestingly, they found that having 5 features in the first convolutional layer and 50 features in the second convolutional layer was more or less optimal.

"Convolutional neural networks have been proposed for visual tasks for many years [LeCun 1998], yet have not been popular in the engineering community. We believe that is due to the complexity of implementing the convolutional neural networks."

They point out that implementation is complicated by the fact that not every unit has the same number of outgoing connections.

The results suggest substantial improvements from both distortions, and the use of convolutional nets. They achieve a best-possible accuracy of 99.6%, which was apparently a record at the time.

2.11 Hinton, Osindero, and Teh (2006)

A Fast Learning Algorithm for Deep Belief Nets

"Learning is difficult in densely connected, directed belief nets that have many hidden layers because it is difficult to infer the conditional distribution of the hidden activities when given a data vector." I don't know why this is. My impression is that it's easy to at least sample from the distribution of hidden activations. Is that false? Or maybe it's true and it's just the calculation of the distribution that is hard. "Variatonal methods use simple approximations to the true conditional distribution, but the approximations may be poor, especially at the deepest hidden layer, where the prior assumes

independence. Also, variational learning still requires all of the parameters to be learned together and this makes the learning time scale poorly as the number of parameters increase." I don't know what variational learning is.

"The network used to model the join distribution of digit images and digits labels... work in progress has shown that the same learning algorithm can be used if the 'labels' are replaced by a multilayer pathway whose inputs are spectrograms from multiple different speakers saying isolated digist. The network then learns to generate pairs that consist of an image and a spectrogram of the same digit class." Fascinating: in other words, it will associate "9" both with different images of 9, and also with different people saying 9.

Discriminative model: a model which can be used to distinguish the MNIST digits. Generative model: a model which, given a label, can be used to generate an image which in some sense samples from the MNIST distribution.

Generative models seems interesting in part because that's what we do (we can both read and write digits, for instance). Of course, it's not entirely clear how these skills are associated. One can learn to read without also learning to write; there are fine motor skills in the latter that are not all that closely associated to reading.

"There is a fine-tuning algorithm that learns an excellent generative model that outperforms discriminative methods on the MNIST database of hand-written digits." I haven't seen this kind of thing mentioned at all in later work — it's all discriminative.

"The learning algorithm is local. Adjustments to a synapse strength depend on only the states of the presynaptic and postsynaptic neuron." This seems very preferable to gradient descent!

Explaining away: Makes inference difficult in directed belief nets. Basically, we can't figure out what root causes must have been, given only partial evidence.

2.12 Hinton and Salakhutdinov (2006)

(link)

Their RBM uses "symmetrically weighted connections". It is not clear to me what this means. It seems to mean that the biases are the same on hidden and visible units. I don't see how that can be — aren't there different numbers of such units?

So the idea is to take an RBM, and then use the training data to find a new set of features. We then use the features generated by the training data as a *new* set of training data, for another RBM. We use that to find new features. And so on, through multiple levels of RBMs. We then use backpropagation to fine-tune the whole thing. It appears that the backpropagation is done with the weights treated as though in a deterministic neural network, not stochastic, as in an RBM.

In a bit more detail, when working with real-valued data, the visible units in later RBMs were set to the activation probabilities of previous hidden units. I.e., probabilities became data.

H and S used a deep network with 784-400-200-100-50-25-6 units. That is, they reduced 784-dimensional input data to just 6 parameters. And, visually at least, their reconstructions were very good, significantly better than 6-parameter PCA and similar techniques.

What makes it difficult to train deep neural networks? I must admit, I don't really have a great answer to this question. Can we come up with a good *a priori* reason for thinking it will be tough? It's not obvious that it should be tougher than a shallow network with the same number of neurons.

H and S compare to the work of Tenenbaum *et al* and Roweis and Saul, and comment: "Unlike nonparametric methods (cites), autoencoders give mappings in both directions between the data and code spaces, and they can be applied to very large data sets because both the pretraining and the fine-tuning scale linearly in time and space with the number of training cases." I don't quite understand the comment about mappings in both directions — I thought the earlier work provided such mappings. Perhaps I should look closer.

2.13 Bengio, Lamblin, Popovici, Larochelle (2007)

Greedy Layer-Wise Training of Deep Networks

They have a complexity-theoretic point of view, a point of view that says depth (in circuits, or otherwise) helps compute functions. I guess this is more or less the point of view of computer scientists who believe that **NC** is a strict subset of **P**.

In general, this is a point of view I haven't much engaged with. I've

been thinking more in the detailed world of the practitioner, wondering just how well a given network functions, and not thinking about these structural questions. But I suppose there is a deep structural question here, which is whether there are deep networks that can compute functions using polynomially many elements, and said functions require exponentially many more elements in a shallow network?

A skeptical way of looking at this is to say that this is a question about scaling, and that scaling isn't what matters for solving pattern recognition problems in the real world, since we have just one such world, of fixed size. But to be skeptical of the skeptic, we would still find it interesting if, in the real world, we were trying to learn functions which were much easier to compute by a deep network than a shallow.

Why might deep networks be better? Two broad reasons: ease of computation; and ease of learning. I'd like to understand both these: Why might computation be easier? And why might learning be easier?

Well, those notes get me to the end of the first sentence of the abstract! Let me skip ahead and see if I can sum up the first paragraph, since it seems very interesting. The basic problem is the ability of various machine-learning algorithms to learn highly-varying functions, "e.g., they would require a large number of pieces to be well represented by a piecewise-linear approximation. Since the number of pieces can be made to grow exponentially... If the shapes of all these pieces are unrelated, one needs enough examples for each piece in order to generalize properly. However, if these shapes are related and can be predicted from each other, 'non-local' learning algorithms have the potential to generalize to pieces not covered by the training set." I can sort of see this: basically, linear boundaries aren't going to give us very much, even with new features: they can't go a huge amount beyond what is already in the input data. But I don't quite see what non-linearities do to get beyond this. I guess it's that we're starting to learn from multiple pieces of training data at once, and making higher-order generalizations. (Basically, once you can do AND gates, you can do conditional logic, and that lets you build up hierarchical reasoning.)

2.14 Pinto, Cox and DiCarlo (2008)

Why is Real-World Visual Object Recognition Hard?

"[W]e show that a simple V1-like [computational?] model — a neurosci-

entist's 'null' model, which should perform poorly at real-world visual object recognition tasks — outperforms state-of-the-art object recognition systems (biologically inspired and otherwise) on a standard, ostensibly natural image recognition test." I'm not sure what moral to take away. That simple systems can do well recognizing natural images? But they also created another "simple" test which demonstrated the inadequacy of their system. "Taken together, these results demonstrate that tests based on uncontrolled natural images can be seriously misleading..." The ultimate conclusion is that they want more focus on real-world image variation, by which they mean that the same object can cast a potentially infinite number of variations on the eye.

"[I]t is not clear to what extent such 'natural' image tests [like Caltech101] actually engage the core problem of object recognition. Specifically, while the Caltech101 set certainly contains a large number of images (9,144 images), variations in object view, position, size, etc., between and within object category are poorly defined and are not varied systematically [I can see that this might be a problem if the sampling is not reasonably fair]. Furthermore, image backgrounds strongly covary with object category [wow!].... The majority of images are also 'composed' photographs, in that a human decided how the shot should be framed [!], and thus the placement of objects within the image is not random and the set may not properly reflect the variation found in the real world. Furthermore, if the Caltech101 object recognition task is hard, it is not easy to know what makesi it hard—different kinds of variation (view, lighting, exemplar, etc) are all inextricably linked together."

"We built a very basic representation inspired by known properties of V1 'simple' cells... The responses of these cells to visual stimuli are well-described by a spatial linear filter, resembling a Gabor wavelet... with a nonlinear output function... and some local normalization (roughly analogous to 'contrast gain control')."

2.15 Deng (2009)

"ImageNet: A Large-Scale Hierarchical Image Database"

An "ontology of images builts upon the backbone of the WordNet structure". "Each meaningful concept in WordNet, possibly described by multiple words or word phrases, is called a 'synonym set' or 'synset'." There are apparently 80 thousand (noun) synsets in WordNet. (I presume there are verb synsets, and perhaps other types as well?) The idea of ImageNet is to

provide 500-1,000 images per synset. That's a grand total of some tens of millions of images. "Images of each concept are quality-cntrolled and human-annotated". To some extent this means that they don't match what we'll actually find "in the wild". This paper reports early work — 5,247 synsets and 3.2 million images.

"ImageNet aims to provide the most comprehensive and diverse coverage of the image world. The current 12 subtrees consist of a total of 3.2 million cleanly annotated images spread over 5,247 categories... To our knowledge this is already the largest clean [what does this mean?] image dataset available to the vision research community, in terms of the total number of images, number of images per category as well as the number of categories." "... to our knowledge no existing vision dataset offers images of 147 dog categories."

Even at very low levels in the tree, ImageNet labels were found to be highly accurate by an independent group of subjects.

"ImageNet is constructed with the goal that objects in images should have variable appearances, positions, view points, poses as well as background clutteer and occlusions." They do an interesting thing to measure diversity. They compute an "average image" for each synset, and then measure the JPG file size. The idea is that very different images will blur out (and so have small file sizes), while more similar images will not (and so will have large file sizes). They find that their images are much more diverse than Caltech101.

Images are collected by querying several image search engines with the appropriate noun or noun phrase. "To obtain as many images as possible, we expand the query set by appending the queries with the word [?] from parent synsets, if the same word appears in the gloss of the target synset [?]". "To further enlarge and diversify the candidate pool, we translate the queries into other languages".

Using Mechanical Turk to label: "In each of our labeling tasks, we present the users with a set of candidate images and the definition of the target synset... We then ask users to verify whether each image contains objects of the synset. We encourage users to select images regardless of occlusions, number of objects and clutter in the scene to ensure diversity." Of course, the problems are that people make mistakes, and they may not agree with one another. They get multiple people to label each image, and only classify something positively if an image gets a convincing majority of the votes. "... different categories require different levels of consensus among users." Basically, the more contentious, the more votes we need to be sure. They

have to do some initial setup to figure out the appropriate thresholds (or if a threshold fails to exist).

Nice idea: classifying at each node in the WordNet net. This reduces the classification difficulty at each step. I wonder if there's a natural way this can be done in deep neural nets? Maybe by building a feature representation unsupervised, and then using those features to train a (tree-like) classifier? "At nearly all levels, the performance of the tree-max classifier is consistently higher than the independent classifier."

2.16 Jarrett (2009)

"What is the Best Multi-Stage Architecture for Object Recognition?": link

This is a much more conventional paper about object recognition than the material I've been reading. The basic idea is to build a pretty good feature extractor, and then to use a standard (supervised) classifier.

"We show that using non-linearities that include rectification and local contrast normalization is the single most important ingredient for good accuracy on object recognition benchmarks." "[T]he SIFT operator applies oriented edge filters to a small patch and determines the dominant orientation through a winner-take-all operation." "Several recognition architectures use a single stage of such features followed by a supervised classifier."

"At first glance, one may think that training a complete system in a purely supervised manner (using gradient descent) is bound to fail on dataset with small number of labeled samples such as Caltech-101, because the number of parameters greatly outstrips the number of samples. [Yes, one might think this] One may also think that the filters need to be carefully hand-picked (or trained) to produce good performance [Yes, at least to the training part], and that the details of the non-linearity play a somewhat secondary role [Again, agreed]. These intutitions, as it turns out, are wrong. [!]"

"A common choice for the filter bank of the first stage is Gabor Wavelets. [A linear filter used for edge detection. Apparently there are similar things in the visual cortex!] Other proposals use simple oriented edge detection filters such as gradient operators, including SIFT, and HoG. Another set of methods learn the filters by adapting them to the statistics of the input data with unsupervised learning. [This is the deep neural nets approach] ... The advantage of learning methods is that they provide a way to learn the filters in subsequent stages of the feature hierarchy. While prior knowledge about

image statistics point to the usefulness of oriented edge detectors at the first stage, there is no similar prior knowledge that would allow to design sensible filters for the second stage in the hierarchy. Hence the second stage *must be learned*." This seems overly pessimistic to me: one can certainly imagine a theory that tells us what features there should be at the second level. Still, it's obviously an attractive model.

"The second ingredient of a feature extraction system is the non-linearity." I don't really understand deeply why non-linearity is so necessary. It'd be good to do so.

Notes that pooling can be applied over space, over scale and space (rescaling?), and over similar feature types and space. "This layer [pooling] builds robustness by computing an average or a max of the filter responses within the pool."

Caltech 101: 101 categories. About 50 images per category, and the size of each image is roughly 300 by 200 pixels. SIFT features plus a linear classifier will give us 50 percent classification accuracy. Using a better classifer will give us 65 percent. "[T]he best results on Caltech-101 have been obtained by combining a large number of different feature families [29]". Reference is to Varma and Ray.

"The hierarchy stacks one or several feature extraction stages, each of which consists of filter bank layer, non-linear transformation layers [*sic*?], and a pooling layer that combines filter responses over local neighborhoods using an average or max operation, thereby achieving invariance to small distortions."

Conclusions: "[U]sing a rectifying non-linearity is the single most important factor in improving the performance of a recognition system[!]" I don't understand the heuristic justifications they give. "Also introducing a local normalization layer improves the performance. It appears to make supervised learning considerably faster, perhaps because all variables have similar variances (akin to the advantages introduced by whitening and other decorrelation methods)."

2.17 Lee (2009) - video

link A video version of the paper below. "We are interested in scaling up deep belief networks to learn generative models and to perform inference on challenging problems." RBMs. Visible nodes: input (training) data.

Hidden nodes: encode statistical relationships in the visible nodes. "Unsupervised training using Contrastive Divergence approximation to maximum likelihood". Deep belief network: "Greedy layerwise training using RBMs". Want to scale DBNs to realistic image sizes: 200 by 200 pixels. One way to deal with this is to use a convolutional net. Alternate between "detection" and "pooling" layers. "Detection layers involve weights shared between all image locations": we have a window of features, sliding across the input image. "Each pooling unit computes the maximum of the activation of several detection units". It shrinks the representation in higher layers. They define a convolutional RBM. It's very similar to a standard RBM, but with a couple of differences. One, the weights are shared across hidden units, as in a convolutional net. Second, they impose a constrain on the hidden units — basically, local sums can't be too large. It's not quite clear to me why they're doing this, but they are. They can still do block Gibbs sampling.

Convolutional DBNs: They do greedy, layerwise training, training one convolutional RBM at a time. They can both infer forwards and backwards through the layers.

Results (MNIST): They trained a two-layer CDBN on *unlabeled* MNIST data. The first layer learns "strokes", while the second layer learns groupings of strokes. Nice results: down to 0.82% error rate. I like the fact that they talk about how the error rate scales with the number of labeled examples.

Results (natural images): First layer it learns localized, orinted edges. Second layer: contours, corners, arcs, surface boundaries. Caltech 101: 65.4% accuracy. Final result is competitive. Training images unrelated to Caltech 101. Three-layer network from faces: first layer learns edges, second layer learns eyes, third layer learns faces. They're computing some kind of precision-recall curve. I don't quite get this — it's an unfamiliar useage to me. They do some training with multiple classes (cars, faces, motorbikes, aeroplanes). The first layer gets general-purpose features. The second layer gets object-class-specific features, as well as some shared features. The third layer gets highly specific features. Nice conditional entropy graph: uncertainty in the class, given the number of features which are active. Wonderful "filling in" of faces.

2.18 Lee (2009)

RBMs. Two layer. Bipartite. Undirected. Binary hidden units, h. Binary or real-valued visible units, v. A weight matrix W between the two layers. If visible units are binary, then we define the energy:

$$E = v^T W h - b^T h - c^T v, \tag{2.8}$$

where b are the hidden unit biases, and c are the visible unit biases. For real-valued visible units, modify the energy by adding a $1/2v^2$ term. This model is simple enough. How should we think about it? The idea is to start with a given set of values for one layer, say the visible layer. Then sample the hidden units. Then sample the visible layer. And so on, ping-ponging back and forth.

"In principle, the RBM parameters can be optimized by performing stochastic gradient ascent on the log-likelihood of the training data." The parameters to be optimized are presumably the weights and biases. The likelihood is the probability of the observed outcomes (i.e., the training data), given the particular parameters. I assume that the idea is that the visible units are supposed to represent the observed data. So we want to choose the parameters of the model in order to maximize the probability of seeing the training data in the visible units. Apparently contrastive divergence is a technique for computing the gradient of the log-likelihood.

Convolutional RBM. The weights between the hidden and visible layers are shared among all locations in an image. What exactly does this mean? Suppose we have an $N_V \times N_V$ image. Then the input layer apparently consists of $N_V \times N_V$ binary units. There are K groups in the hidden layer, each an $N_H \times N_H$ array of binary units. So there are $N_H^2 K$ total hidden units.

We index the hidden groups by k. Each hidden group has a bias, b_k. All visible units share a single bias, c.

For any given group, k, we have a single set of $N_W \times N_W$ weights (the "filter"). $N_W \equiv N_V - N_H + 1$. The basic idea is to filter the inputs, but translating the filter across the input image.

I will come back to the energy function a little later. XXX. We can do Gibbs sampling to generate the appropriate distributions.

2.19 Scherer (2010)

Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition

Notes that many standard models are based on Hubel and Wiesel: the Neocognitron, convolutional nets, HoG, SIFT, Gist features, and HMAX. "These models can be broadly distinguished by the operation that summarizes over a spatial neighbourhood. Most earlier models perform a subsampling operation, where the average over all input values is propagated to the next layer... A different approach is to compute the maximum value in a neighborhood... While entire models have been extensively compared, there has been no research evaluating the choice of the aggregation function so far. The aim of our work is therefore to empirically determine which of the established aggregation functions is more suitable for vision tasks. Additionally, we investigate if ideas from signal processing, such as overlapping receptive fields and window functions can improve recognition performance."

They note that there are so many variants on complex cells / pooling operations that it's impossible to do a complete analysis. Instead, they're going to choose a particular model and analyse that, based on convolutional neural networks. "Our choice of a CNN is largely motivated by the fact that the operation performed by pooling layers is easily interchangeable without modifications to the architecture."

"The purpose of the pooling layers is to achieve spatial invariance by reducing the resolution of the feature maps." Is that really right? We don't actually want spatial invariance — relative positions matter. But the details don't matter. It's a way of saying small spatial shifts (relative to feature size) don't matter. So a better sentence would be: the purpose of the pooling layers is to ensure that small spatial shifts (relative to feature size) don't matter.

"We evaluate two different pooling operations: max pooling and subsampling." Subsampling computes an average and multiples by a trainable scalar. Max pooling applies a window function and computers the maximum in the neighbourhood.

They wanted to do the following: (1) figure out how max pooling and subsampling compare; (2) determine whether overlapping pooling windows improve performance; and (3) find suitable window functions.

"For both NORB and Caltech-101 our results indicate that architectures with a max pooling operation converge considerably faster than those employing a subsampling operation. Furthermore, they seem to be superior in

selecting invariant features and improve generalization." Of course, this conclusion only applies in their specific context. Maybe if we increased the data set then this would no longer be true? Or if we changed the architecture in some other way? Furthermore, no explanation of why it is true has been given.

"To evaluate how the step size of overlapping pooling windows affects recognition rates, we essentially used the same architectures as in the previous section. Adjusting the step size does, however, change the size of the feature maps [I don't see why — we can make them the same size] and with it the total number of trainable parameters, as well as the ratio between fully connected weights and shared weights." I must admit I don't understand what's being done here, or in the remainder of this section. I think it would be dangerous for me to take much away from it.

Comparison to Coates' (2011) paper: Coates found that shorter stride length helped. However, the model used seems to have been quite a bit different to this paper. So I'm not sure I'd read too much into either result — more study is, I think, needed, to understand this.

2.20 Coates (2011)

An Analysis of Single-Layer Networks in Unsupervised Feature Learning

"In this paper... we show that several simple factors, such as the number of hidden nodes in the model, may be more important to achieving high performance than the learning algorithm or the depth of the model... Our results show that large numbers of hidden nodes and dense feature extraction are critical to achieving high performance." They actually get state-of-the-art performance using only a single layer of features. This is interesting: it's a case where deep learning *doesn't* help. But increasing the number of features *does* help — a lot!

Reviews the standard practice: use unsupervised learning to pre-train multiple layers of features.

"Even with very simple algorithms and a single layer of features, it is possible to achieve state-of-the-art performance by focusing effort on these choices [number of features, dense feature extraction, whitening] rather than on the learning system itself."

"[W]e employ very *simple* learning algorithms and then more carefully choose the network parameters in search of higher performance. If (as is

often the case) larger representations perform better, then we can leverage the speed and simplicity of these learning algorithms to use larger representations."

CIFAR-10: 60,000 32 by 32 colour images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images. CIFAR-10 is a subset of the "80 million tiny images" dataset.

CIFAR-100: Like CIFAR-10, but with 100 classes containing 600 images each. I.e., CIFAR-100 is a more difficult problem.

So the CIFAR data sets can be thought of as small but challenging class recognition data sets.

"It will turn out that whitening, large numbers of features, and small stride lead to uniformaly better performance regardless of the choice of unsupervised learning algorithm... the main contribution of our work is in demonstrating that these considerations may, in fact, be *critical* to the success of feature learning algorithms — potentially more important even than the choice of unsupervised learning algorithm. Indeed, it will be shown that when we push these parameters to their limits that we can achieve state-of-the-art performance, outperforming many other more complex algorithms on the same tasks."

This really makes me wonder about the standard claims made about deep learning.

"Since the introduction of unsupervised pre-training, many new schemes for stacking layers of features to build 'deep' representations have been proposed. Most have focused on creating new training algorithms to build single-layer models that are composed to build deeper structures. Among the algorithms considered in the literature are [long list]. Thus, amongst the many components of feature learning architectures, the unsupervised learning module appears to be the most heavily scrutinized."

"Some work, however, has considered the impact of other choices in these feature learning systems, especially the choice of network architecture. Jarret et al. [11], for instance, have considered the impact of changes to the "pooling" strategies frequently employed between layers of features, as well as different forms of normalization and rectification between layers." One reason this is interesting is that it suggests a direction in which to take work.

"While we confirm that some feature-learning schemes are better than others, we also show that the differences can often be outweighted by other factors, such as the number of features. Thus, even though more complex learning schemes may improve performance slightly, these advantages can be

overcome by fast, simple learning algorithms that are able to handle larger networks." [It'd be nice to know more about the impact of changed data set size as well.] Summing up: more sophisticated algorithms may not be as useful as increasing the basic parameters in a simple algorithm. But given this, I'd like to know why Ng used a deep RICA network in his later work?

"At a high-level [*sic*], our system performs the following steps to learn a feature representation: 1. Extract random patches from unlabeled training images. 2. Apply a pre-processing stage to the patches. 3. Learn a feature-mapping using an unsupervised learning algorithm. [So this is how we learn the features to be used. Now we move to classification.] Given the learned feature mapping and a set of labeled training images we can then perform feature extraction and classification: 1. Extract features from equally spaced sub-patches [why equally spaced? why use sub-patches?] covering the input image. 2. Pool features together over regions of the input image to reduce the number of feature values. [I guess this makes sense if we're using small local features, as does the use of sub-patches.] 3. Train a linear classifier to predict the labels given the feature vectors."

"It is common practice to perform several simple normalization steps before attempting to generate features from data. In this work, we assume that every patch $x^{(i)}$ is normalized by subtracting the mean and dividing by the standard deviation of its elements. For visual data, this corresponds to local brightness and contrast normalization."

"For our purposes, we will view an unsupervised learning algorithm as a 'black box' that takes the [training] dataset X and outputs a function $f : R^N \rightarrow R^K$ that maps an input vector $x^{(i)}$ to a new feature vector of K features, where K is a parameter of the algorithm."

After learning features, they do a type of convolutional extraction: basically, stepping across the images with a particular stride length, and extracting K-dimensional features at each stage.

They do a funny form of pooling. They split their features up into four quadrants, and simply sum over each quadrant. That gives them a total of $4K$ features to use for classification. I must admit, this seems to me like a rather strange procedure to use. They don't appear to discuss it at much length.

After pooling they use a linear classifier — an SVM, with the regularization parameter determined by cross-validation.

"For sparse autoencoders and RBMs, the effect of whitening is somewhat ambiguous. When using only 100 features, there is a significant benefit of

whitening for sparse RBMs, but this advantage disappears with larger numbers of features. For the clustering algorithms, however, we see that whitening is a crucial pre-process since the clustering algorithms cannot handle the correlations in the data."

Whitening made a big difference for both k-means measures, and for Gaussian mixture models. It made only a small difference for the sparse autoencoder and for the RBM.

The number of features made a big difference for all approaches. It's not clear what the asymptotic performance will be, but even with 1600 features (where they stopped) things were still improving quite a bit.

The stride length also had a huge impact on performance. I find this really interesting! It'd be interesting to understand the performance tradeoffs.

Size of the local receptive field didn't have quite as much of an impact. Indeed, increasing the size sometimes decreased performance, when other factors (e.g., number of features) was held constant.

They got the best known results on CIFAR 10 using k-means. (Note that this has since been greatly improved.)

"Our results above may seem inexplicable considering the simplicy of the system — it is not clear, on first inspection, exactly what in our experiments allows us to achieve such high performance compared to prior work.... Each of the network parameters (feature count, stride and receptive field size) we've tested potentially confers a significant benefit on performance. For instance, lare numbers of features (regardless of how they're trained) gives us many non-linear projections of the data... using extremely large numbers of non-linear projections can make data closer to linearly separable and thus easier to classy. [E.g., the kernel trick] Hence, larger numbers of features may be uniformly beneficial, regardless of the training algorithm"

"It appears that large receptive fields result in a space that is simply too large to cover effectively with a small number of nonlinear features."

Takeaways: the notion of a pipeline: feature learning by unsupervised techniques, followed by a standard classifier (e.g., SVM); increasing the number of features learned can help *a lot*; larger local receptive fields don't seem to help, and can actually hinder; a shorter stride length can help quite a bit; K-means (using the triangle technique) can help a lot.

2.21 Le, Karpenko et al (2011)

"ICA with Reconstruction Cost for Efficient Overcomplete Feature Learning": link

ICA as a technique for unsupervised feature learning. Point out that standard ICA learns orthonormal features, while they want overcomplete feature sets. "Using our method to learn highly overcomplete sparse features and tiled convolutional neural networks, we obtain competitive performances on a wide variety of object recognition tasks. We achieve state-of-the-art test accuracies on the STL-10 and Hollywood2 datasets."

"Sparsity has been shown to work well for learning feature representations that are robust for object recognition." What exactly is a sparse feature? I guess in the case of sparse autoencoders we only allow a relatively small number of hidden neurons to be on. Algorithms for learning sparse features: sparse auto-encoders, RBMs, sparse coding, and ICA.

"[Standard] ICA has two major drabacks. First, it is difficult to learn *overcomplete feature representations*". Goes on to claim that classification performance works better when features are overcomplete. This makes a certain amount of sense: certainly, there should be no problem having overlapping features. Also claims that ICA is sensitive to whitening, and this makes it difficult to scale ICA to high dimensional data.

Regular ICA: Let x^j be training data. Choose a penalty function $g(\cdot)$. They suggest $g(z) = \log(\cosh(z))$. Let W_j be a row in a weight matrix. Then $W_j x^k$ measures the overlap between the weight vector and the training data. If it's one, then x^k is very much like the weight vector. And if it's less than one, then it's less so. So we simply sum over features, W_j, and over training data, x^k. The goal is to "find the best features", i.e., to minimize:

$$\sum_{jk} g(W_j x^k). \tag{2.9}$$

This is done subject to the constraint that $WW^T = I$, i.e., the feature vector are orthonormal to one another. ICA is done assuming zero mean for the training data, $\sum_k x^k = 0$, and unit covariance, $\sum_k x_k x_k^T = mI$. This is achieved by whitening the data.

Reconstruction ICA (RICA): Minimize:

$$\frac{\lambda}{m} \sum_k \|W^T W x^k - x^k\|^2 + \sum_{jk} g(W_j x^k). \tag{2.10}$$

In other words, find the features which minimize the cost, while preserving the training data pretty well. "We use the term 'reconstruction cost' for this smooth pentalty because it corresponds to the reconstruction cost of a linear autoencoder, where the encoding weights and decoding weights are tied". Note that tying is not used in the LRM paper. This makes it more similar to a standard autoencoder, as I've described elsewhere in my book.

"ICA's main distinction compared to sparse coding and autoencoders is its use of the hard orthonormality constraint in lieu of reconstruction costs." The basic idea in proving some kind of equivalence is to let λ be large. "If the data is whitened, RICA is equivalent to ICA for undercomplete representations and λ approaching infinity."

From my point of view, the main thing here is simply the basic problem formulation: the function to minimize. I'd like to think of this in a slightly more connectionist fashion. Let me think back to the cost function. Minimizing the first part means that we have weights which allow us to approximately reconstruct the training data. Minimizing the second part is more an l1 constraint, roughly speaking it's telling us to have few features. So we have features which let us reconstruct, and we are likely to have only a few features at a time.

Local receptive field TICA: "[L]ocal receptive field neural networks are faster to optimize than their fully connected counterparts [because they have fewer parameters]. A major drawback of this approach, however, is the difficulty in enforcing orthogonality across partially overlappling patches. [This becomes a severe constraint if we only overlap at a few points.] We show that swapping out locally enforced orthogonality constraints with a global reconstruction cost solves this issue. [I.e., we can forget about local orthogonality, and just worry about optimizing the cost.]" It seems that they do this by minimizing the following function:

$$\sum_k \|W^T W x^k - x^k\|^2 + \sum_{jk} \sqrt{\epsilon + H_j (W x^k)^2}. \tag{2.11}$$

A few things: (1) λ should presumably appear out the front of the first term; (2) They never explain ϵ; (3) The H_j are pooling matrices; (4) It's not clear what $(Wx^k)^2$ means — presumably the elementwise square; (5) I don't see how $H_j(Wx^k)^2$ can be a scalar.

2.22 Tenenbaum, Kemp, Griffiths, and Goodman (2011)

(link) A review of a particular approach to inductive learning. They want to combine Bayesian learning with complex ways of representing knowledge.

Claims that there is strong evidence that children can learn to generalize their use of words from just a few examples. This suggests that there must be some pretty clever underlying patterns to how we generalize. "A massive mismatch looms between information coming in through our senses and the outputs of cognition".

Claims that we humans do reason (implicitly) in Bayesian ways about a number of things. Mostly omits the evidence that we *don't* in some important ways. This omission bugs me. They *do* mention the fact that our conscious assessements of probability tend to be terrible, which is pleasing. With that said, I'm not certain about this — I just have the strong impression that there are well-known instances where we certainly don't reason in a Bayesian way. It'd be good to have references.

"The biggest remaining obstacle is to understand how structured symbolic knowledge can be represented in neural circuits." Interesting. I've often wondered exactly this. They make the followup comment: "Connectionist models sidestep these challenges by denying that brains actually encode such rich knowledge". That seems too strong to me, but there is some truth to it: the connectionists seem less interested than one might suppose in this question, perhaps believing that its solution should be deferred.

How would one go about solving this problem? Actually, what would a solution / better statement of the problem even look like? Maybe we could encode entry-relationships? In particular, let us suppose we want to encode XYZ where X and Z are entities, and Y is the relationship. One way of encoding this would be to have a neural network with nodes for each entity and for each relationship. We'd try to design the network so that the only relationships which are active would be those which are true, given the active entities.

2.23 Bengio (2012)

(link)

Notes that many of the recommendations haven't been proved, they're heuristics that have emerged out of experimentation. "A good indication of the need for such validation is that different researchers and research groups do not always agree on the practice of training neural networks".

Claims that the optimal learning rate is usually close to the largest learning rate that does not cause divergence of the cost function. Heuristic: start with a large learning rate, and if the cost function increases, start again with a training criterion that is three times smaller.

This can be automated by keeping track of the cost from epoch to epoch. If the cost got *larger* during an epoch, then decrease the training rate by a factor two, say. If the cost got *smaller*, then increase the training rate by a factor of 1.1, say. How well will that work? I worry that we'll end up with a situation where we're mostly going back and forth between the training rate being too high, and too low, with not enough time to really learn anything.

Larger mini-batches allow a modest increase in learning rate. I don't understand the details of this. It'd be nice to have some heuristics. Large mini-batches will certainly reduce stochastic error from the sampling. Is that what's going on? Or is there some other reason?

"Because the gradient direction is not quite the right direction of descent, there is no point in spending a lot of computation to estimate it precisely for gradient descent." In other words, do frequent rapid estimates rather than slow accurate computations.

It seems to me that it'd be helpful to keep track of training examples with markedly different gradients. Those are ones which we could learn a lot from. There's an idea here, which is to *identify outliers* using the gradient. We should oversample from the outliers. I'll bet that improves performance, if the right oversampling rate is chosen. I've explored this idea further below.

Bengio confirms that for large data sets, mini-batch stochastic gradient descent is pretty much non-optional.

The use of validation data to train hyper-learners, which learn hyper-parameters for a learning algorithm.

Comments that the initial learning rate is often the single most important hyper-parameter. "If there is only time to optimize one hyper-parameter and one uses stochastic gradient descent, then this is the hyper-parameter that is worth tuning." Also comments that there's often little benefit to doing anything other than keeping the learning rate constant. When doing otherwise, Bengio suggests a strategy of keeping the learning rate constant for the first τ steps, and then decreasing it as $1/t$, where t is the number of

steps. Note that this strategy is not the same as the (exponential) automated strategy I describe above. Suggests setting τ by waiting until the cost goes up. Also suggests setting multiple values for the schedule, and seeing how they compare.

Mini-batch size: between 1 and a few hundreds. Typical value of 32. Notes that this mostly affects computation time, not the final value of the cost.

Number of epochs: Watch the validation error, and stop once we're beginning to overfit.

Momentum: smooth out gradient by taking an average of recent gradients.

Comments that increasing the number of hidden neurons in all layers results in a quadratic increase in time. It's not clear to me why that should be the case — obviously there is a quadratic increase in the number of weights, and so a quadratic increase in time per epoch. But maybe it'll take a larger number of epochs to converge?

"[W]e found that using the same size for all layers worked generally better or the same as using a decreasing size (pyramid-like) or increasing size (upside down pyramid), but of course this may be data dependent."

I am surprised by this. It seems to contradict our ideas about feature learning. It'd be good to look at Larochelle et al's results. Perhaps it reflects the fact that *more* high level concepts can be formed out of the "atoms" of input than there are atoms.

"For most tasks that we worked on, find that an overcomplete first hidden layer works better than an undercomplete one."

It's not really clear why this is the case. Again, it may be that it's because there are more high-level concepts than low level one. Still, that seems to be at odds with my intuition about autoencoders.

States that this is particularly true for unsupervised learning. That *is* consistent with the idea that it's because there are many different abstractions possible, far more than basic features.

Claims that there is a "clean Bayesian justification" for regularization as the negative log-prior. The discussion that follows is extremely interesting and I'm still sorting it out. The picture that emerges seems to be that what we're doing when learning is using some kind of maximum likelihood estimation. In particular, we start with some sort of prior in parameter space — a Gaussian — and then try to find the weights maximizing the probability of the parameters (weights), given the training data. I need to unpack this

still further: it's regularization as a form of maximum likelihood. For now I'll proceed, and then return to this later.

Normalization: Claims that we should normalize the regularization parameter by B/T, where B is the mini-batch size, and T is the number of training examples. This is consonant with what I've observed.

Early stopping and L2 regularization: comments that these two are essentially equivalent, and that one may as well drop L2 rgularization when engaged in early stopping. I don't believe this. The solution spaces will be completely different in the two cases. I'm happy to believe that *sometimes* they'll give the same result, but see no reason to believe that they'll always give the same outcome.

L1 regularization and feature selection: Comments that this strongly suppresses irrelevant weights. Also comments that you may wish to consider doing both L1 and L2 regularization, with different regularization parameters. That seems sensible to me.

Q: An alternative approach to choosing λ is to regard it as an extra parameter beyond the weights, and to apply gradient descent to it as well. How well would this work? My first instinct is to think that it won't work — that λ will be driven to zero. But upon more reflection things are more complicated than that. It'd be interesting to know.

Sparsity: Increase sparsity can be compensated by a larger number of hidden units. A sparsity-inducing penalty can be viewed as a way of regularizing. Note that it's no longer so easy to view this in the Bayesian framework. Notes that the L1 penalty seems most natural, but is not often used. Try to push the (mini-batch) average to a particular constant.

Neuron nonlinearity: Bengio notes that he's most often used the sigmoid, the tanh, $\max(0, a)$, and the hard tanh. Interesting remark about the sigmoid not working well as the top layer of a deep supervised net without unsupervised pretraining. Apparently it's okay for auto-encoders.

Weight initialization: Sample uniformly on $4\sqrt{6/(\text{fan-in} + \text{fan-out})}$. This will give us a total length equal to roughly the number of layers.

Hyper-parameter selection as an optimization problem: points out the dangers of overfitting your validation data.

Q: When does it make sense to say that we're overfitting?

Approach to parameter search: doing it logarithmically.

Q: Does it make sense to do gradient descent on just a subset of weights at a time? I do wonder if that wouldn't sometimes yield better results. Deep

learning has something of this flavour.

2.24 Bengio 2012

Bengio, Courville, and Vincent: "Representation Learning: A Review and New Perspectives": http://arxiv.org/pdf/1206.5538v2.pdf.

"This paper reviews recent work in the area of unsupervised feature learning and joint training of deep learning, covering advances in probabilistic models, auto-encoders, manifold learning, and deep architectures." "... much of the actual effort in deploying machine learning algorithms goes into the design of preprocessing pipelines and data transformations that result in a representation of the data than can support effective machine learning." While I know this last is true, I haven't actually had to do a whole lot of data cleaning myself, yet. "What makes one representation better than another? Given an example, how should we compute its representation, i.e. perform feature extraction? Also, what are appropriate objectives for learning good representations?

"Speech was one of the early applications of neural networks, in particular convolutional (or time-delay) neural networks... Microsoft has released in 2012 a new version of their MAVIS... speech system based on deep learning".

"Transfer learning is the ability of a learning algorithm to exploit commonalities between different learning tasks in order to share statistical strength, and *transfer knowledge* across tasks." There are apparently competitions for transfer learning. I wonder what sorts of problems are being attacked? "Of course, the case of jointly predicting outputs for many tasks or classes, i.e., performing *multi-task* learning also enhances the advantages of representation learning algorithms"

"Unfortunately,... most of these algorithms [SVM etc] only exploit the principle of *local generalization*... they rely on examples to *explicitly map out the wrinkles of the target function*. Generalization is mostly achieved by a form of local interpolation between neighboring training examples... We advocate learning algorithms that are flexible and non-parametric, but do not rely exclusively on the smoothness assumption. Instead, we propose to incorporate generic priors such as those enumerated above into representation-learning algorithms." This starts to get at the point of view that says that neural network architecture is all about figuring out how we generalize. If there is a hierarchical structure in how to generalize well, that's what your

network will need. If not, it won't. "Kernel machines are useful, but they depend on a prior definition of a suitable similarity metric, or a feature space in which naive similarity metrics suffice. We would like to use the data, along with very generic priors, to discovery these features, or equivalently, a similarity function."

They make a really nice point about expressiveness. "[H]ow many parameters does [a model[require compared to the number of input regions (or configurations) it can distinguish?" They argue that a deep net can distinguish exponentially more regions than more conventional approaches.

2.25 Bottou (2012)

(link)

Notes that there are theorems about the convergence time for batch gradient descent (time is logarithmic in the eventual error), and for second-order gradient descent. It's really not clear how valuable such results are; I guess it's comforting that they exist.

Notes that there are some powerful results about the convergence of stochastic gradient descent, under conditions like $\sum \eta^2 < \infty, \sum \eta = \infty$. Apparently the "Robbins-Siegmund theorem" helps with convergence. The relevant paper is here.

Monitor both the training cost and the validation error: Suggests periodically evaluating the validation error during training, and stopping training when it hasn't improved after some time.

2.26 Ciresan (2012)

link This uses just straight-up backprop to train a neural net — no convolutional nets, no pretraining, just online learning with backprop. The main tricks are to use numerous deformed training images, and graphics cards to speed up learning. Apparently, Simard et al used a single hidden layer with 800 neurons to get an accuracy of 99.3 percent on MNIST. (It'd be interesting to know whether they deformed the images?)

The paper asks whether it was really true that the pre-training is necessary? Can't you just train for a long time? And the answer seems to be yes!

They train online, using slightly deformed images, and claim that this means they can use the whole MNIST set for validation. This seems suspect to me — it relies on the deformations being more or less independent of how the network generalizes. Let's run with it, however.

They trained 5 networks, with 2 to 9 hidden layers each. From 1.34 to 12.11 million free parameters. They have a variable learning rate that shrinks by a constant factor after each epoch, from 0.001 down to 0.000001. This seems absolutely crucial to their success. I'm a little surprised by the use of the constant factor decrease, since that will bound the "total" (so to speak) learning distance travered, simply because the geometric sum converges. It seems like you'd get better performance if you chose a learning schedule where terms decreased more slowly, so the sum of the learning rates diverged. That's true of the hyperbolic function advocated by Bengio in his 2012 paper, whose sum will diverge (albeit, only logarithmically). They initialized weights uniformly at random in the range -0.05 to 0.05 — that's close to, but not the same as, the $1/\sqrt{fan-in}$ that I've preferred. They use a tanh activation function.

They used a GPU to do computations. It apparently sped the deformation routine up by a factor of 10, and forwardprop and backprop by a factor of 40! That's a big improvement.

Typical architecture: 784-1000-500-10 neurons. They get 0.44 percent test error. That's pretty close to perfect. The most complex architectures were: 784-2500-2000-1500-1000-500-10 and 784-9 x 10000-10. These get test errors of 0.32 and 0.43 percent, respectively. Interestingly, there seems be some advantages to having non-homogeneous numbers in the layers.

Took 93 CPU seconds to deform the MNIST images. 87 of those seconds were for the elastic distortions, so that's what they converted to the GPU. When doing the conversion they converted MNIST images to 29 x 29 to get a proper center, which simplifies distortion.

2.27 Domingos (2012)

link

He points out that we don't have access to the function we really want to optimize, unlike in most optimization problems. Instead we use training error as a proxy for test error. That's a very interesting and strange situation.

"Learners combine knowledge with data to grow programs."

Overfitting has many faces: "the bugbear of machine learning"; "it comes in many forms that are not immediately obvious". Generalization error can be decomposed into bias and variance. Bias is the tendency to keep learn the same wrong things. Variance is the tendency to learn random things. E.g., an SVM (without kernel) may have high bias if the data is nowhere close to linearly separable. Cross-validation can itself start to overfit.

Intuition fails in high dimensions: I don't think this is quite right. It would be better to say that it needs to be replaced in high dimensions.

Theoretical guarantees are not what they seem: Points out that there are effectively guarantees that can (with caveats) be put on induction. Very interesting. It'd be good to understand this in conjunction with the no-free lunch theorems.

Feature engineering is the key: Points out that the "machine learning" part of a machine learning project may be tiny. More time spent gathering data, cleaning it, and figuring out good input features.

More data beats a cleverer algorithm: "As a rule, it pays to try the simplest learners first". "... the organizations that make the most of machine learning are those that have in place an infrastructure that makes experimenting with many different learners, data sources and learning problems easy and effcient, and where there is a close collaboration between machine learning experts and application domain ones."

Representable does not imply learnable: in other words, don't focus all your attention on one representation (say, neural nets, or SVMs) merely because there is some kind of universality theorem for them.

Correlation does not imply causation: Keep it in mind when interpreting the results of machine learning algorithsm.

2.28 Hinton 2012 — Coursera

Lecture 5 b: Object recognition: If you want to solve computer vision, it may help to find features that are invariant under things like rotation, translation, and so on. Example: parallel lines with a red dot between them. This is invariant under rotation and translation, but may actually be quite a useful feature. I guess I can imagine similar features being use to recognize an eye. Relationship between features may themselves be captured by other features. The idea of normalizing an image: once normalized, it may be easier to extract features. Of course, that then requires us to solve the problem:

how to normalize? (Hinton claims, without presenting anything so gauche as actual evidence, that we don't mentally rotate images to recognize them.) One approach to normalization: brute force approach, trying all possible boxes, in a wide range of positions and scales.

Lecture 5c: Convolutional neural networks for handwriting recognition: Early example of deep neural nets, from the 1980s. The idea is to *replicate features*. So an edge is a good feature — and if it's a good feature at one point in the visual field, then it's probably a good feature at other points in the visual field. Put another way, a feature detector that's useful at some point in the visual field is likely to be useful elsewhere, too. Replication across position reduces the number of parameters to be learned. It's easy to learn replicated features with backpropagation. I guess we just constrain the weights to be the same. So we want $\Delta w_1 = \Delta w_2$. We just average the gradients across partial derivatives. An advantage is that if we can learn to detect a feature in one place, then we learn how to detect it in other places. Hinton advocates against rotational or scale invariance. I don't know if that's a good idea, frankly — it seems to me that with modern computers that may be practical. The idea of pooling adjacent replicated features. Hinton advocates either averaging or the max (he says max is a little better). LeNet was used to read something like 10 percent of all checks in North America, according to Hinton. There's still a frontier associated to MNIST, and it may be worth trying to push that frontier. The idea of generating synthetic data (in part to reduce overfitting). McNemar test.

Lecture 5d: Convolutional neural networks for object recognition: Apparently most people doing vision with neural nets have switched to using rectified linear activation function, not just a sigma function. A good paper on this appears to be "Deep Sparse Rectifier Neural Networks" (Bengio et al). Use left-right reflection of images to get more training data. And use image subsets to get more training data. Uses GPUs: 500 cores per GPU, very fast at matrix-by-matrix arithmetic, very high bandwidth to memory.

Lecture 6a: stochastic mini-batch gradient descent: Hinton calls this the most frequently used algorithm for training neural networks. He says it's often preferable even to techniques from the optimization community. How to choose a learning rate: if the error keeps getting worse or oscillates wildly, reduce the learning rate. If the error is falling slowly, increase the learning rate. Do this all automatically.

Lecture 15a: PCA: Lots of data in a very high-dimensional space. But

maybe there's a low-dimensional manifold on which most of the data lies. In some sense that manifold captures much of the structure in the data. What we want is a projector onto a lower-dimensional subspace. Suppose $x_1, x_2, \ldots, x_m$ are our data points. Obvious idea is to stick .

2.29 Hinton (2012) - videos

IPAM Summer School videos

Attributes backprop to Paul Werbos. It was done in his 1974 PhD thesis. Hinton also lists several others, including Amari, Parker, and LeCun. Points out that deep learning didn't work well, except in time delay and convolutional networks. Says that part of the reason Werbos was ignored was because he was applying backprop to econometrics, where it was hard to see the value.

Why deep learning is feasible today: He starts with simple raw speed, not pre-training, interestingly enough. He also says that there's been a "small improvement in the theory". Says the biggest disappointment with backprop was that it didn't work with recurrent neural nets. I don't understand how this squares with Williams and Zipser.

"On the whole backpropagation fell out of favour because it failed to be able to learn multiple layers of features." Says that convolutional nets were the only ones where deep learning worked.

"Almost everything I used to believe about backpropagation is wrong."

"What is wrong with back-propagation? It requires labeled training data. [Well, no, not if you use ideas like autoencoders.]"

Why is the learning time slow in deep nets? If you use the right scales for the weights, you can do some of this learning much faster.

He's strongly emphasizing the unsupervised learning / feature learning point of view. Basically, pretraining to initialize, and then fine-tuning with labelled data and backprop.

"You can get a lot of knowledge into the network by messing with the training data." Analogizes to education.

On the advantages of generative models: learn $p(\text{image})$, not $p(\text{label}|\text{image})$. "If you want to do computer vision, first learn computer graphics." I think that overstates the case, but there's something to it. Reminiscent of the idea that learning is really memory.

Belief nets: A directed acyclic graph of stochastic variables. We learn

the values of some variables. We'd like to infer the states of the other variables. And we'd like to adjust the interactions between variables to make the network more likely to generate the observed data. (Obviously, this is all very close to Pearl's causal models.)

Neat point about stochastic models: it lowers the communication cost in distributed models. Send 1 bit instead of 32 or 64 bit float.

Points out that while it's easy to generate examples at the leaf nodes, it's hard to infer causes. Yet that's exactly what we want to do.

Suppose we observe some output data. Let's suppose we can sample the hidden states in an unbiased fashion. Now update the weight by $\Delta w_{ji} = \eta s_j(s_i - p_i)$. Note that here, j is a parent, and i is a child node. This is more or less Hebb's rule. "Nice local learning rule".

Monte Carlo methods: painfully slow for large, deep models. Can be used to sample from the posterior. Variational methods are much faster. You get the wrong result, but it's bounded away from the right result. "Inferring the wrong posterior and then doing learning anyway".

RBMs: The feature detectors are genuinely independent (given the data). The posterior distribution is easy to sample from. The partition function makes learning difficult. He derives a nice quick way to learn an RBM: $\Delta w_{ij} = \eta *$ a difference of averages of correlations between visible and hidden units.

2.30 Krizhevsky (2012)

link 1.2 million images in ImageNet 2010. 1000 classes. 650,000 neurons. Five convolutional layers. Max pooling layers. Three fully-connected layers. 1000-way softmax. Used dropout to prevent overfitting.

Past image data sets: NORB, Caltech-101/256. CIFAR-10/100. "Simple recognition tasks can be solved quite well with datasets of this size, especially if they are augmented with label-preserving transformations." "But objects in realistic settings exhibit considerable variability, so to learn to recognize them it is necessary to use much larger training sets". LabelMe: hundreds of thousands of fully-segmented images. ImageNet: 15 million labeled high-res images in over 22,000 categories.

This paper: trained a very large convolutional neural net on subsets of ImageNet used in two competitions. Got by far the best results ever reported on those data sets. Removing any convolutional layer significantly decreased

performance. "All of our experiments suggest that our results can be improved simply by waiting for faster GPUs and bigger datasets to become available."

ImageNet: 15 million images, 22,000 categories. ILSVRC: 1000 images in 1000 categories. 1.2 million training images, 50,000 validation images, and 150,000 testing images. ILSVRC-2010: test set labels are available. Top-5 error rate: the fraction of test images for which the correct label is not among the five labels considered most probable by the model.

ImageNet has variable-resolution images. They down-sampled to 256×256. They did this by rescaling the image so the shorter side was of length 256. Then cropped out the central 256×256 patch. They also subtracted the mean activity over the training set from each pixel. This was the complete pre-processing.

Architecture: 8 layers. 5 convolutional. 3 fully-connected.

ReLU Nonlinearity: Instead of sigmoid function they used $f(z) = \max(z, 0)$. They refer to this as a *rectified linear* unit. "Deep convolutional neural networks with ReLUs train several times faster than their equivalents with tanh units". I believe it is standard wisdom that convolutional nets work better with tanh units than sigmoid. "The magnitude of the effect [faster learning...] varies with network architecture, but networks with ReLUs consistently learn several times faster than equivalents with saturating neurons".

Training on multiple GPUs: Done in part because the training set wouldn't fit into a single GPU's memory.

Local response normalization: They do a local normalization step, essentially a kind of brightness normalization. It reduces error rates by a little over 1 percent.

Overlapping pooling: Again, a slight improvement.

Architecture: The first convolutional layer filters the 224 by 224 by 3 image with 96 kernels of size 11 by 11 by 3. There is a stride distance of 4, i.e., the distance between the receptive field centers of neighbouring neurons. I need to understand quite a bit more about CNNs and pooling.

Lots of overfitting: 1.2 million examples, 10 bits of info per example (1 in 1000 classification). But 60 million parameters. So overfitting is a real problem.

Data augmentation: (1) image translations and horizontal reflections. Extracting 224 by 224 patches. This gives them a factor 2048 more training data. The network makes a prediction by extracting five 224 by 224 patches and their horizontal reflections, and averaging the predictions made by the

network's softmax layer. (2) Altering the intensities of the RGB channels in the training images. Perform PCA on ImageNet and use it to modify the images. "This scheme approximately captures an important property of natural images, namely, that object identity is invariant to changes in the intensity and color of the illumination."

Dropout: "a very efficient version of model combination that only cost about a factor of two during training". Set to zero the output of each hidden neuron with probability 0.5. Don't contribute to forwardprop nor to backprop. Every time the network is trained it has a different architecture, but the architectures share weights. "This technique reduces complex co-adaptation so neurons, since a neuron cannot rely on the presence of other neurons". This is going to be useful in very large networks with a relative paucity of data. "Without dropout, our network exhibits substantial overfitting. Dropout roughly doubles the number of iterations required to converge."

Used SGD with momentum. "We used an equal learning rate for all layers, which we adjusted manually throughout training. The heuristic which we followed was to divide the learning rate by 10 when the validation error rate stopped improving with the current learning rate. The learning rate was initialized at 0.01 and reduced three times prior to termination." That seems like a useful heuristic.

Results: top-1 test set error rate: 37.5 percent. top-5 test set error rate: 17.0 percent. That seems incredibly good, although not human comparable. They also report a bunch of other results: every single one is very, very good.

2.31 Le (2012)

Building high-level features using large-scale unsupervised learning

I have described the architecture elsewhere. Let me describe some of the results. They took 13,026 faces from Labeled Faces in The Wild, and about 24,000 distractor objects from ImageNet. I'm not especially keen on this procedure — it seems like there might be very easy ways to distinguish the two data sets that have little to do with whether or not a face is present.

"After training, we used this test set to measure the performance of each neuron in classifying faces against distractors. For each neuron, we found its maximum and minimum activation thresholds, then picked 20 equally spaced thresholds in between. The reported accuracy is the best classification

accuracy among 20 thresholds." There's a lot that's not being said here. Which neurons are we considering? Every neuron in the network? Or just in the last layer? And it's not actually stated that higher activations are the right criterion (as opposed to lower). However, I think I can reasonably infer that's the case, because of the pooling.

"The best neuron in the network achieves 81.7 percent accuracy in detecting faces". That's compared to a guessing strategy, which achieves 64.8 percent. They found that removing the local contrast normalization reduced this number to 78.5 percent.

The performed a numerical optimization to find the optimal stimulus. This really is quite striking: it's definitely a face!

They did a very interesting control experiment, removing the faces from the unlabelled training data, using OpenCV. The recognition accuracy of the best neuron dropped to 72.5 percent. So it's not just that we're detecting ImageNet versus Labeled Faces in the Wild.

Invariance properties: They used 10 face images and did some scaling, rotation, x and y translations. The face feature detector still worked pretty well for rotation, not so well for the other operations. Still, it's interesting that this is possible at all, especially since rotation and scaling are going to be hard to build into the network.

Other feature detectors: They repeated the face detector work, but with cats and human body parts. They got 74.8 % and 76.7 %, respectively. The data sets were constructed so that random guessing would give 64.8 percent, as for the faces. They also tried some deep autoencoder experiments, and found that while there were selective neurons in those networks, they weren't nearly as good as with the muli-RICA-layer architecture.

ImageNet: On the 2011 data set — 16 million images, 20,000 categories — they achieved 15.8 percent accuracy, a huge jump over the best (9.3 percent) results.

2.32 Le (2012), video

"Tera-scale deep learning": link

Problems with standard hand-crafted features: the features may not generalize to another domain; the features take a long time to develop. (Recall Hinton: the time of hand-engineered features is over.) "We're still stuck at SIFT and HOG".

RICA: Built on TICA (topographic independent component analysis). We have some data, x^i. Take a 3 by

RICA can learn from any (unlabelled) data. Can learn features from videos to do action recognition. E.g. "Get out of car". "Eat". And so on. Very interesting features! SIFT / HoG is also used for video, apparently.

Four most famous activity recognition data sets: KTH. Hollywood2. UCF. YouTube. They outperform SIFT / HoG on four best-known data sets.

On cancer / MRI: our usual intuition breaks down. It really helps to be able to automatically discover features.

"Scaling up deep RICA": This is the way to think about what the Google-Stanford paper does.

"Using a thousand machines alone is not enough." They needed to change their algorithms.

"Higher layer [i.e., later features] are very difficult to visualize." I don't understand why that is the case.

Two main ideas in scaling up: local connectivity; asynchronous SGD.

1 billion parameters. I wonder how they avoid overfitting?

They pick out a neuron in the top layer. They look to see: which images in the test (?) set stimulate that neuron the most? And then they do a numerical optimization to figure out what the optimal input stimulus is.

Classify "sting ray" versus "manta ray" in ImageNet.

2.33 Ng (2012) — video

Ng's contribution to 2012 IPAM workshop

"Instead of doing AI, we ended up spending our lives doing curve fitting."

Has a nice example of building a motorcycle recognizer by building a feature classifier to determine whether there are wheels or handlebars, and then adding an extra classifier layer. There's actually a general story here: we can recursively do this.

He makes stronger claims about the brain rewiring (e.g., in ferrets) than I've heard before. Says that it's been done in four animal species. (I wasn't immediately able to find this using Google Scholar, it'd be interesting to see sources.) Also says that it's vision in every sense that he understands what vision means. I must admit I didn't get that out of Sur's original paper —

it seemed like a much coarser sense of vision. It'd be interesting to know if finer-grained tests have since been done.

"The complexity of the trained algorithm comes from the data, not the algorithm."

Distinguishes semi-supervised versus self-taught learning (unsupervised feature learning). The problem with semi-supervised learning is that it requires some broad constraints on classses — e.g., we need all images to be of cars or motorcycles. Self-taught learning is much easier.

Sparse coding: Learns a dictionary of basis functions so that each training image can be decomposed sparsely in terms of basis functions. (Use a "sparsity penalty term".) If you train sparse coding on natural images you get edge detectors. "It's more useful to know where the edges are in an image than where the pixels are." "It gives us an alternate way of representing the image." "ICA version of spare coding." Recursively do sparse coding.

Sparse deep belief network (Honglak Lee). One layer: edges. Second layers: models of object parts. Third layers: object models.

Ng believes it is mostly scalability that is the issue. More features, more data =¿ better results. What appears superficially to be an algorithmic superiority is really about the availablity of more data, or more computational power that allows more features to be learned.

Should we use higher-order algorithms? Conjugate gradient? L-BFGS. Gradient descent with line search. Black-box algorithms which will take the gradient and cost and just work. Ng's favourite: L-BFGS.

"The most reliable indicator of whether [a new grad student] has got gradient descent to work is whether they do gradient checking." The problem: buggy implementations will learn. Just not as well as a correct implementation.

2.34 Sutskever (2013)

On the importance of initialization and momentum in deep learning

"Deep and recurrent neural networks... are poweful models that were considered to be almost impossible to train using stochastic gradient descent with momentum. In this paper, we show that when sotchastic gradient descent with momentum uses a well-designed random initialization and a particular type of slowly increasing schedule for the momentum parameters, it can train both... to levels of performance that were previously achiev-

able only with Hessian-Free optimization. We find that both the initialization and the momentum are crucial since poorly initialized networks cannot be trained with momentum and well-initialized networks perform markedly worse when the momentum is absent or poorly tuned. Our success training these models suggests that previous attempts to train deep and recurrent neural networks from random initializations have likely failed due to poor initialization schemes." In other words, we can train deep neural nets with (momentum-based) stochastic gradient descent, *provided* we're careful about how we initialize the weights, and provided we do the appropriate things with the momentum.

"Martens (2010) attracted considerable attention by showing that... Hessian-free Optimization... is capable of training [deep neural nets] from certain random initializations without the use of pre-training, and can achieve lower errors for the various auto-encoding tasks considered by Hinton and Salakhutdinov (2006)."

The picture that is starting to appear: Overall achievement = Quality of algorithm + quantity of data + number of features + amount of computing time.

"The first contribution of this paper is a much more thorough investigation of the difficulty of training deep and temporal networks than has been previously done... We show that while a definite performance gap seems to exist between plain SGD and HF on certain deep and temporal learning problems, this gap can be eliminated or nearly eliminated... by careful use of classical momentum methods or Nesterov's accelerated gradient."

Apparently Polyak introduced the momentum technique, and obtained some results on how much faster they can be than position-based techniques.

NAG: Nesterov's Accelerated Gradient. "[F]or general smooth (non-strongly) convex functions and a deterministic gradient, NAG achieves a gloabl convergence rate of $O(1/T^2)$ (versus the $O(1/T)$ of gradient descent), with constant proportional to the Lipschitz coefficient of the derivative and the squared Euclidean distance to the solution." I don't know what T is here. NAG turns out to be a variation on the momentum method, with the only difference being that we compute the gradient at the updated position. "While the classical convergence theories for both methods [NAG and momentum] rely on noiseless gradient estimates (i.e., not stochastic), with some care in practice they are both applicable to the stochastic setting." "However, the theory predicts that any advantages in terms of asymptotic local rate of convergence will be lost... a result also confirmed in experiments...

For these reasons, interest in momentum methods diminished after they had received substantial attention in the 90's. And because of this apparenty incompatability with stochastic optimization, some authors even discourage using momentum or downplay its potential advantages"

The key point seems to be to separate out two timescales. One is the initial transient phase, when we're still hopping between regions of different local minima, before the phase of fine local convergence. "[I]n ractice, the 'transient phase'... seems to matter a whole lot more for optimizing deep neural networks. In this transient phase of learning, directions of reduction in the objective tend to persist across many successive gradient estimates and are not completely swamped by noise." "Thus, for convex objectives, momentum-based methods will outperform SGD in the early or transient stages of the optimization where L/T is the dominant term." Here, L is the Lipshitz coefficient of the gradient.

Why NAG works: "This benign-looking difference seems to allow NAG to change v in a quicker and more responsive way, letting it behave more stably than CM [classical momentum[in many situations, especially for higher values of μ. Indeed, consider the situation where the addition of μv_t results in an immediate undesirable increase in the objective f. The gradient correction to the velocity v_t is computed at position $\theta_t + \mu v_t$ and if μv_t is indeed a poor update, then [the gradient at the new position] will point back toward θ_t more strangly than [the gradient at the old position], thus providing a larger and more timely correction to v_t than CM." I don't think this is quite right. It's not a question of pointing back to the original position. It's a question of pointing in the right direction, which may be different than the direction of the original position. Still, this line of reasoning otherwise seems sound. (And it's probably true in two dimensions.)

"While each iteration of NAG may only be slightly more effective than CM at correcting a large and inappropriate velocity, this difference in effectiveness may compound as the algorithms iterate." I don't know that this is the case. It seems more likely to me that rather than accumulating small improvements, it actually is preventing occasional bad mistakes.

There is a nice example in the appendix. Basically, a 2d example, with very elongated ellipses as the contours. The momentum method (with low friction) has the problem that it only very slowly builds up momentum in the right direction. Basically it overshoots early on, and then has to swing backwards and forwards, slowly. NAG avoids this, even though it also has low values of momentum.

They analyse CM and NAG for the objective function $C(y) = \sum_j \lambda_j y_j^2 + c_j y_j$. In this particular case, they prove that NAG acts like the classical momentum technique with learning rate η, but with a modified momentum $\mu(1 - \eta\lambda_j)$ in component j. It should be possible to prove this through a straightforward computation.

This means that for small learning rates CM and NAG become very similar. Note that locally every (smooth, convex) cost function may be approximated by a quadratic cost functional (i.e., approximating by the appropriate quadratic locally), and so this behaviour is likely generic. We also see that NAG is going to have lower momentums, and so more friction; it will tend to damp out oscillations. The decrease in momentum will be particularly high when λ_j is large. This is good: it decreases the momentum a lot, and so increases the friction a lot, which damps out the overoscillation that will cause a problem as we go through the $y_j = 0$.

Takeaway technique here: to understand an optimization technique it can really help to look at quadratic trial functions like this, where it may be possible to analyse behaviour analytically. In particular, the big benefit of analysing quadratic cost functions is that they really do carry most of the (local) information we'll ever need.

"The aim of our experiments is three-fold. First, to investigate the attainable performance of stochastic momentum methods on deep autoencoders starting from well-designed random initializations; second, to explore the importance and effect of the schedule for the momentum parameter μ assuming an optimal fixed choice of the learning rate...; and third, to compare the performance of NAG versus CM."

They don't look at test errors — i.e., they ignore regularization and overfitting. Not sure how bulletproof their argument for this is — it seems a bit like special pleading. But I'm happy to run with it: the point is that optimization can be treated separately from generalization. (Of course, it may be that a better optimization method is worse at generalization, and that ultimately needs to be looked into.)

The schedule for μ which they used was to look for the smaller of $\mu_{\max}$ and $1 - 1/2(\lfloor t/250 \rfloor + 1)$, where t is the epoch number. In other words, over the first 250 iterations, $\mu = 1/2$. Then we switch to 3/4. Then to 5/6. And so on, until we get to $\mu_{\max}$. They have a nice explanation for this. Basically, use the $1 - 1/t$ type schedule when the function isn't convex — this will helps us explore and gradually find a good locality. But once that is found it's better to switch to a constant rate, which will converge exponentially

quickly. So this is a nice hybrid.

Actually, that's not quite the full schedule. It turns out that they do a final modification of μ for the tail end of training, reducing it to another constant. They have a nice heuristic explanation for this, which I won't get into here, but should perhaps come back to in future.

The results are impressive: much, much better than basic stochastic gradient descent.

They also investigate recurrent neural networks. I'm less familiar with these, so I'll just quickly write out some very telegraphic and incomplete notes. Echo-state networks. "ESNs ... have achieved high performance on tasks with long range dependencies (?)" "RNNs were believed to be almost impossible to successfully train on such datasets [with long-range temporal dependencies], due to various difficulties such as vanishing/exploding gradients" Interesting comments on the spectral radius of the hidden-to-hidden matrix on the dynamics of a RNN. When are RNNs likely to be useful? "The main achievement of these results is a demonstration of the ability of momentum methods to cope with long-range temporal dependency training tasks to a level which seems sufficient for most practical purposes." In practice, of course, many (most?) interesting human cognitive tasks involve long-range temporal dependency: I do action X now, then must do Y later, than Z (which depends on X) later still, and so on. RNNs seem like they might be especially useful for "chains of thought" as opposed to pattern recognition.

Comparison to HF: They note that HF is a truncated Newton method. Sounds like it's an improved linear conjugate gradient method. "[Conjugate gradient] accumulates information as it iterates which allows it to be optimal in a much stronger sense that any other first-order method (like NAG)"

The idea behind all these first-order methods — momentum-based and NAG — seeems to be to find indirect ways of putting curvature into the problem, by computing gradients at two separate points. This gives us information about the locally approximating quadratic, rather than the locally approximating plane. It seems as though you could do better by making use of even more points — three or four would give you higher-order still approximations. (They still wouldn't give you global information, though.)

I need to get clear on the relationship of curvature to gradient descent. The basic point is that if the cost surface is highly curved in some direction, gradient descent will tend to send us in that direction. That's not always what we want. Sometimes we want to move off along low curvature directions as well. That's typically the case for a general (positive-definite) quadratic.

2.35 Summary of CIFAR-10 results

As at July, 2013. I have drawn heavily on the compendium of results by Rodrigo Benenson. Note that CIFAR-10 contains 10 classes, with 5,000 training images per class, and 1,000 test images per class. Images are 32 by 32, and in RGB. Not centred or size-normalized.

Note that the accounts below are not at all complete, they are intended as a quick first cut. Several of these should be investigated in much more depth.

Karpathy on CIFAR: http://karpathy.ca/myblog/2011/04/27/lessons-learned-from-manually-classifying-cifar-10-with-code/

Snoek, Larochelle, and Adams (2012) (link): "Practical Bayesian Optimization of Machine Learning Algorithms". Appears to provide the best results at the time of writing. They used a three-layer convolutional neural network. Achieved an error on the test set of 14.98%. This is over 3% better than state of the art (without augmenting the data). They then augmented the data using horizontal reflections and translations, getting the error down to 9.5 % on the test set.

An interesting aspect of the project is that they learnt the hyper-parameters automatically. In particular, they did a Bayesian optimization to learn 9 separate hyper-parameters, including the number of epochs, the learning rate, and the width, scale and power of the response normalization in the pooling layers. The learned hyper-parameters significantly outperform a human expert's optimization of the hyper-parameters. Their expert achieved 18% and 11% error (without and with data augmentation, respectively).

Code from this project is available. Note that Jasper Snoek is at U of T, but will be leaving for Harvard in September. They based their convolutional net implementation on cuda-convnet.

Krizhevsky, Sutskever, and Hinton (2012): (link) "ImageNet Classification with Deep Convolutional Neural Networks" A four-layer convolutional neural net achieved 13% test error rate without local response normalization, and 11% with local response normalization. Used cuda-convnet.

Ciresan, Meier, and Schmidhuber (2012): (link) "Multi-column deep neural networks for image classification" Achieves 11.21% error for CIFAR-10. Achieves 0.23 % error for MNIST. Claims that humans get a 0.2% error, with citation (would be interesting to look up). Use a deep convolutional network. They do basic backprop, with no pretraining. The architecture is to repeat a convolutional layer followed by max pooling multi-

ple times, followed by some fully connected layers. They use 2 by 2 receptive fields and max-pooling regions. It appears that the stride length is 2, as well. Somewhat similar to Krizhevsky et al's ImageNet paper. They use a fully online training algorithm. They use a GPU. They use what they call a multi-column deep neural network, which I don't quite understand — looks to be a technique for training multiple networks and combining the results. They used a (scaled) tanh function for convolutional and fully connected layers, a linear activation function (does this mean rectified?) for max-pooling layers, and softmax at the output. They used online gradient descent, with an annealed learning rate (0.001, decaying by a factor of 0.993 after every epoch), and continual translations, scaling and rotation of images. Initial weights are drawn from a uniform random distribution in the range [-0.05, 0.05].

MNIST architecture: 29 by 29 input; a 20-map convolutional layer, with a receptive field of 4 by 4; max-pooling of 2 by 2 regions; a 40-map convolutional layer with 5 by 5 receptive field; max-pooling of 3 by 3 regions; fully connected layer with 150 neurons; fully connected (softmax) layer with 10 neurons.

CIFAR architecture: 3 by 32 by 32 input; 300-map convolutional layer, with 3 by 3 receptive fields; max-pooling of 2 by 2 regions; 300-map convolutional layer, with 2 by 2 receptive fields; max-pooling of 2 by 2 regions; 300 convolutional maps, 2 by 2 receptive fields; max-pooling of 2 by 2 regions; then fully connected layers with 300, 100 and 10 neurons.

Augmenting the training set (by translating up to 5%) helps a lot. Scaling (up to 15 percent), rotation (up to 5 degrees) and additional translations (up to 15 percent) helps a little extra.

The contrast between the Krizhevsky and Ciresan results suggests that ideas like dropout and rectified linear units make a big difference.

Q: How much difference does the larger number of maps in the convolutional layers make?

Tentative conclusions: Use, in roughly this order: Martens' initialization; rectified linear units; dropout; augmented training data; annealed learning rate. It'd be interesting to look at the local contrast normalization. Also try looking at Nesterov's momentum method. The Ciresan results suggest some benefit from using lots of maps in the convolutional layers.

2.36 Grandmother cell (Wikipedia)

Apparently proposed in the late 1960s by Konorski and Lettvin. Lettvin "originated the term grandmother cell to illustrate the logical inconsistency of the concept." There is apparently quite a bit of support for the concept at the broad category level: neurons which are higly face-specific, and even to individual human faces. However, "[e]ven the most selective face cells usually also disharge, if more weakly, to a variety of individual faces." A 2005 study found a "neuron for Halle Berry", which fired not only for pictures of the actress, but also to the words "Halle Berry", and which didn't fire when pictures of several other actresses were presented. Of course, this doesn't mean that was the only cell to respond. The "sparseness" hypothesis versus the "distributed representation" theory. It's really not clear to me that there is a dichotomy here. A picture of Halle Berry will no doubt cause many neurons to fire, some of which will fire for other reasons too. Maybe the hypothesis is this: for each single object or concept there is a corresponding grandmother neuron.

Chapter 3

Miscellanea

Compiling to neural networks: Can we create compilers which translate programs written in a conventional programming language into a neural network? I'd be especially interested in seeing how this works for AI workhorses such as Prolog. What could we learn from such a procedure? (1) Perhaps we could figure out how to link up multiple neural modules, with one or more of the modules coming from the compiler? (2) Maybe we could use a learning technique to further improve the performance of the compiled network. Googling doesn't reveal a whole lot, although I did find a paper by Thrun where he discusses decompiling, i.e., extracting rules from a neural network. Thrun uses a technique he calls validity-interval analysis, basically propagating intervals for inputs and outputs forwards and backwards through a network.

Deep learning requires nonlinear neurons: Put another way, deep learning with linear neurons doesn't help. Via linear embedding it's equivalent to a single hidden layer whose size is just the minimal size of any of the original hidden layers. So there is absolutely no advantage to doing deep learning with linear neurons.

No theory of generalization: We have all these techniques based on parameter-fitting. But we have a paucity of strong underlying theoretical ideas.

Principal Components Analysis (PCA): It'll be useful to review PCA here. Suppose we have a set of data points x in some high-dimensional (vector) space. Then we'd like to find a k-dimensional projector P such that

the following error function is minimized:

$$\sum_x \|x - Px\|^2. \tag{3.1}$$

This error can be rewritten as $\mathrm{tr}((I - P)\Sigma)$, where $\Sigma \equiv \sum_x xx^T$. And so we simply choose P to project onto the eigenvectors of Σ with the k largest eigenvalues. The *principal components* are the eigenvectors of Σ, in order of decreasing eigenvalue. (There may, of course, be some ambiguity when Σ is degenerate).

Practically speaking, suppose we have a billion images, each of which can be regarded as a vector in a 100,000-dimensional space. We can reduce to (say) a 100-dimensional space. This gets rid of much of the irrelevant structure, and hopefully leaves a structure that is useful for comparing images.

PCA and autoencoders: PCA is a way of simplifying our understanding of data in high dimensions. Think of the space of all possible images. There's a subset of that space which can plausibly be taken to represent faces. (Note that contextual clues can also help). How can we characterize that subspace? Classic example of PCA: IQ testing. Take a large number of different tests. Turns out that there is a common factor. Another nice example: a helix in 3 dimensions. There's a major question: how to determine the number of hidden units?

Recurrent neural networks (RNN): According to Wikipedia, RNNs have achieved the best results to date on handwriting recognition. An obvious question is: what are the respective advantages of RNNs and feedforward networks? Are there important problems for which one or the other is preferable? Why? What I've read about these questions is opaque.

Regularization: I'd like to understand *why* we regularize. Certainly, regularization results in solutions with a small norm. But why do we not what solutions with a larger norm? Will something bad happen to us if we allow such solutions?

The standard argument: what's bad is that overfitting can occur. And thus regularization helps reduce overfitting. It'd be nice to have an example where overfitting actually occurs. It's really not clear that there *should* be a problem with overfitting. In fact, neural networks eventually become virtually invariant under rescaling of their weights and biases. So it's really not clear that it should help.

Returning to regularization, here's the standard story people tell to explain why they regularize. The story is that they want to avoid high-

complexity solutions, in order to avoid over-fitting. Solutions with smaller norms are in some sense lower complexity. And therefore it makes sense to look for solutions with smaller norm. One way of doing this is to penalize solutions with larger norms. Thus, we should add a term to the cost which penalizes such solutions.

Now, this is just a story. It's not in any sense a sharp justification. In fact, the impact of regularization is still being understood. Researchers write papers where they try different approaches to regularization, compare them to see which works better, and try to understand why different approaches work the way the day.

When can overfitting occur? Typically, when there are more parameters in the model than there is training data. What's odd about this is that regularization doesn't really help all that much with this problem. It just restricts one degee of freedom.

Many different types of regularization possible. I will just use the most standard and obvious, which is quadratic. Anything which penalizes high-complexity solutions is okay. It's really a research topic.

Empirically: I find that regularization seems to help. When we regularize I get higher accuracies, by quite a bit. I don't understand why that is.

Maybe I'm already overfitting, and regularization is helping reduce that problem. It's possible: I have 20,000 or so parameters in my model. It'd be nice to see if this is the case.

An example of overfitting: I'll bet I can it to overfit when we use just 50 training examples. And I can probably more or less prove this using cross-validation.

Look at LeCun *et al*'s results: do they regularize, or not?

Restricted Boltzmann machines: The idea is not to learn a function, but rather to learn a probability distribution. There are two layers of neurons: a visible layer, and a hidden layer. All visible units are connected to all hidden units. The energy of a given configuration is just:

$$E(v,h) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_{ij} w_{ij} v_i h_j \tag{3.2}$$

$$= -a \cdot v - b \cdot h - v^T W h \tag{3.3}$$

where a are the biases for the visible units, b are the biases for the hidden units, and W is the weight matrix. The distribution is just the standard Boltzmann distribution, at some fixed temperature. Apparently it can be

shown that:

$$p(v_i = 1|h) = \sigma(a_i + (Wh)i), \tag{3.4}$$

where σ is the usual sigmoid function. (I'll bet this is easy to show, just by summing out all the other visible units.) Furthermore, the v_i are independent of one another, given h. This too would be easy to show — it'll be a straightforward consequence of the bipartite nature of the graph. So we can compute the probability of v, given h, simply by multiplying sigmoids.

Let's suppose we wanted to train an RBM with a set of images. The images would correspond to the visible units, while the hidden units would be feature detectors. The idea is to adjust the weights and biases so that training images have a high probability, i.e., a low energy.

In a little more detail, suppose we input a training image. Then we can stochastically pick a corresponding value for the hidden units. Now, feed that back, and stochastically choose a value for the image. In an ideal world, we'd recover the original image. We modify the weights in such a way as to improve the fidelity of the recovered image.

Well, the penny finally drops: an RBM can be viewed as a neural network in which the transitions are probabilistic. That's all! Frankly, we don't even really need the stuff about ground states, although it's a beautiful thing to keep in mind.

Softmax function: Suppose q_j is some set of values. Then we define the softmax function by:

$$p_j \equiv \exp(q_j)/\sum_k \exp(q_j). \tag{3.5}$$

This is a probability distribution, which preserves the order of the original values. You can, for example, take the softmax in the final layer of a neural network, taking the weighted sum of inputs as the q_j values, and then applying the softmax. The output from the network can then be interpreted as a probability distribution.

Thinking geometrically: Suppose we're asked to tell the difference between pictures of a human face, and pictures of a giraffe. We can represent the pictures as points x in a very high-dimensional space. And so our task is to divide that space up into two parts: one is classified as giraffe, the other as human face. (Maybe it should be three parts: the thrid part would be: neither face nor giraffe). And so what we really want is algorithms for

dividing up that space. In some sense we're interested in understanding the space of all such algorithms.

It'd be interesting to lay out all the different curlicues to thinking in this way: the opportunities, and the pitfalls. There are at least three broad approaches: (1) the *pure geometric approach*, based on finding mathematical structures to divide the space; (2) the *biological approach*, where we try to figure out how we do it; and (3) the *kludge approach*, where we simply try lots of ideas, and pile them up on top of one another. That's a pretty rough division, but seems like a good starting point for thought. My bet is that progress comes from playing these ideas off against one another.

Tricks: Much of what seems to be going on is the discovery of tricks (of various generality) which can be used to improve pattern recognition performance. There are some general heuristics: *use symmetry* is obviously one.

3.1 Future reading

Ciresan 2012 on MNIST, and Rifai 2011 ("The manifold tangent classifier") on MNIST.

Kiros 2013: http://www.ualberta.ca/ř̃kiros/kiros_thesis_jun5.pdf Best reported results on MNIST when no distortions are used

Interesting comments on image recognition: https://news.ycombinator.com/item?id=5994851

Saxe et al: "On random weights and unsupervised feature learning" (2011). On hyper-parameter optimization. One of Ng's collaborators.

LeCun on recent Ng results: https://plus.google.com/104362980539466846301/posts/5ab217Hu

Goodfellow: https://plus.google.com/103174629363045094445/posts/dh7UT9xbMW4

SIFT:

Tips on what works: https://news.ycombinator.com/item?id=5994851

"Fast, accurate detection of 100,000 object classes on a single machine": http://googleresearch.blogspot.ca/2013/06/fast-accurate-detection-of-100000.html

Hinton: "Where do features come from?": http://scholar.google.ca/citations?view_op=view_cita

Bengio lecture notes

Seide 2011 on deep learning and Microsoft's MAVIS system.

Bengio and COurville: "Deep learning of representations" http://www.iro.umontreal.ca/ bengioy/papers/BengioCourvilleChapter.pdf

Andrew Ng, CS294A lecture notes

McCulloch and Pitts

Recent Bengio paper on new approach to deep learning:http://arxiv.org/abs/1306.1091

Elliasmith

Martens 2010: Hessian-Free optimization, and sparse initialization.

Bengio et al "Scaling Learning algorithms towards AI" 2007

Boureau: A theoretical analysis of feature pooling in visual recognition (2010).

Sermanet: Convolutional neural networks applied to house numbers digit classification

Elkan 2013: Learning meanings for sentences: http://cseweb.ucsd.edu/ẽlkan/250B/learningmea

Agre.

Hubel and Wiesel: 1959. Simple and Complex. The basic model of V1.

Frome 2009: Large-scale Privacy Protection in Google Street View

Collobert: "Natural language processing almost from scratch:"

Bengio et al (1994): The vanishing gradient problem. "Learning long-term dependencies with gradient descent is difficult".

Erhan: "Why does unsupervised pre-training help deep learning?"

HoG:

Hinton et al (2006):

Itamar Arel et al For a different POV.

PAC learning.

Conference on learning representations: http://techtalks.tv/iclr2013/

IPAM: https://www.ipam.ucla.edu/schedule.aspx?pc=gss2012

Lee and Mumford (2003): link This looks like great background reading on the idea of doing hierarchical inference in the visual cortex.

Embrechts (2010):

Dropout:

Le (2012): link

Seide (2011): link

Bengio (2007): link

Ranzato (2007):

Lee (2008):

Larochelle (2009):

Wolpert (XXX): No free lunch.

The NIPS 2012 talks:

Elements of statistical learning: link

No more pesky learning rates: link

Olshausen and Field:

Tenenbaum 2011: How to grow a mind

Rumelhart et al on backprop.

BigBrain Atlas: http://news.sciencemag.org/sciencenow/2013/06/bigbrain-atlas-unveiled.html

Hinton on DReDnets: http://techtalks.tv/talks/drednets/58115/

Distributed deep learning: link.

Stanford tutorial: http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial

Eliot R. Smith: "What do connectionism and social psychology off each other?" Good for something of an exterior point of view.

To do: Contrastive divergence (http://learning.cs.toronto.edu/ hinton/absps/cdmiguel.pdf and http://www.cs.utoronto.ca/ hinton/absps/nccd.pdf). LeCun 1998 "Efficient BackProp". Dropout. Maxout. Andrew Ng's 1997 paper "Preventing overfitting of cross-validation data". Blumer *et al* with guarantees on induction: (http://scholar.google.ca/scholar?cluster=11895938102761137877&hl=en&as_sdt=0,5). Would be good to understand this in conjunction with no free lunch. NIPS papers are online.

Neural nets FAQ: No one definition of a neural network. It's possible to do XOR with just a single hidden layer, if direct connections to the output from the input are allowed. Problems which neural nets aren't so good at: predicting random or pseudo-random numbers; factoring large integers; determining whether a number is prime. Research problem: find a net which will determine whether a number is prime. Distinction between recurrent and feedforward neural networks. Calls the set of cases we'd like to generalize to the *population*. Constructive learning: start with a small network, train, then gradually add extra neurons, and do more training. A lot of work has been done on toy problems, and various hacks are known for the different toy problems.

Stephen Judd (1988): Thesis on complexity of learning in neural networks: http://www.dtic.mil/dtic/tr/fulltext/u2/a450825.pdf.

Sima (1996): Shows that finding weights is hard even for sigmoidal neural networks with just 3 nodes. This can be viewed as an extension of Blum and Rivest (1989). http://scholar.google.ca/scholar?cluster=18396613610240979409&hl=en&as_sd

Egri and Schultz: Found a neural network capable of recognizing prime numbers. http://www.cs.mcgill.ca/~legri1/prime06.pdf