1 Yann LeCun

“The interesting problems are nonconvex.”

What’s happened over the last 5 years is a revolution that has infected many areas in computer science: deep learning, neural nets rehashed and modernized, taking advantage of fast computers. There’s been a gold rush of companies using this for speech recognition (the reason you can talk to your smartphone now), image processing, etc. The same thing happened with computer vision in 2013: everyone dropped what they were doing and did deep learning. Industry picked up this much faster than academia: Microsoft, Google, facebook, IBM, etc.—because it works. We don’t understand why it works; we only have intuition and hand-wavy arguments. (Even simpler things than deep learning are nonconvex and we don’t know why they work.)

People upload 2 billion photos to facebook every day (600 million directly to facebook). They go through 2 deep learning systems: face recognition and image understanding. This allows facebook to select content relevant to you. Ex. The Who’s Nearby app instantly identifies your friends when you snap a photo. It’s a convolutional neural network.

Take image, run face detector, identify key points on the face, rectify, normalize (rotate to be frontal, etc., so eyes, etc. are always in the same place), unfold the sides of the face, feed it to a convolutional neural net. It’s been trained to classify thousands of people, and learn a metric. “These people should be close together, these people should be far apart.” It’s really an embedding into Euclidean space. Start with 100×100 pixels, reduce to 4000 dimensions. Use a hashing technique to compress to 256 bits for quick comparison.

Naor: what about embedding before neural nets?

There is no well-defined initial metric to compare faces.

Think of each person is a manifold. The ambient dimension is low, bounded by muscles and lighting sources (forget about hair). How do you separate the manifold?

Train the distance metric (“Siamese neural net”):

$$\|G_v(X') - G_w(X')\|^2 .$$

Train so pictures of the same person are close together, and of different people are far away.

What did people use before deep learning? PCA. Normalize the image and project it onto a low-dimensional space maximizing variance. It doesn’t work that well. People call this “eigenfaces.” Then people used sparse representations, etc. Highly engineered things—hand crafted features like distance between eyes and nose. Deep learning means we don’t have the hand-craft the features.

There is a technique called linear classification.

A linear classifier looks at $\text{sign}(w^T x + b)$. The categories should be linearly separable. Then people used a kernel Φ and calculated $w^T \Phi(x)$. But for say, facial recognition, the number of basis functions is ridiculously large—this method very sensitive to movements in a few pixels, etc. The main problem is invariance. The mapping $G_w(X)$ has to have complicated invariant properties. It has to be smooth: as long as the variability makes you move in a flat manifold you can separate it from others. Variations should move on a flat manifold. If the last step is a dot product, then variations on a flat manifold don’t matter.

The next step is to stack multiple layers of basis functions. Take basis functions from previous layer and combine them nonlinearly.

Take a vector at a layer, multiply by a matrix, take each component and send through nonlinear function (ex. rectification). (We used to use sigmoid functions, but rectification $1_{\geq 0}x$ works better.)

Here's a handwavy argument for using layers. Suppose we're learning boolean functions, ex. most significant bit of sum of the two.

We can do this in two layers: compute the intermediate results. The bottom is $\wedge$, the top is $\vee$. But for most functions, the number of minterms in DNF is exponential. If you allow multiple layers, you allow algorithms with greater steps. Ex. for addition, you can do with a linear number of nodes in a linear number of layers. This took 30 years for the machine learning community to understand.

It's rare that we use a full matrix. We give some structure, ex. for convolutional neural nets. The matrix is a "filter bank."

An image is 3 2-dimensional arrays (RGB). Suppose we have only 1.

1. The first layer generates a set of planes the same size as the original ones. Each is a dot product of a region (5×5 neighborhood) with a weight vector. It's not smoothing/averaging, ex. looking at derivatives/differences.

2. Do subsampling: each plane reduces resolution: each one takes the max over a region. This gives shift invariance: if we shift the original picture by 1 pixel, and subsample in 2×2 squares, then it's shifted by $\frac{1}{2}$ pixels there. The manifold of transforms is more linear; the curvature is smaller.

3. Now repeat 5–7 times, so the resolution shrinks at every steps. At the end we get a vector each component having dimension 1.

The planes are different features, ex. the first layer identifies oriented edges, crosses; the second corners, gratings; the third, circles, squares... The influence area of a particular unit increases in later layers—they give global features of the image.

Figure out all paths from an input to output going through all intermediate notes. It's either active if all the nodes are the linear (not flat) region of the rectifier, or inactive. The contribution is

$$\widetilde{y} = \sum_p \delta(p, w, x) x_p \prod_{i \in P} w_i.$$

x_p is the input taken from path p. $\delta(p)$ indicates whether the path is active. (Here we're just considering rectifiers. Max nodes also activates a switch and is similar.) We want to minimize the loss function. A good one is the hinge loss function. Suppose we want to do binary classification. A training sample is a pair (X^k, Y^k). We want to minimize the loss function

$$L(w) = \frac{1}{Q} \sum_{k=1}^{Q} L(Y^k, \widetilde{Y}(X^k, W)).$$

Often people take

$$L = [m - Y^k \widetilde{Y}(X^k, W)]^+$$

where $[\cdot]^+$ is rectification.

The loss function is

$$L(w) = \frac{1}{Q}\sum_k \sum_p \delta(p, X^k, Y^k) X_p^k \prod_i w_i$$

. (I'm forgetting the structure of the network, except that each path has L components where L is the number of layers.) The same w_i appears in many paths. You're trying to learn a sequence of matrices: the collection of w_i from each layer to the next. It's a piecewise polynomial function whose pieces are very small.

Let's turn ourselves into physicists and do something ridiculous:

1. $\delta(p, X^k, Y^k)$ is a random variable, sometimes 1 vs. 0 independent of the product. (It's actually deterministic.)

2. Every path starts at a different input.

This is an "ideal gas model" of neural nets (particles in gas never touch each—but this gives good results, $PV = nRT$.) But it's not that different. (cf. Belief propagation assuming an infinite tree.)

How do you optimize a function like this? We use stochastic gradient descent. Compute the gradient of the loss (one sample/one term in the sum). We do this efficiently by applying the chain rule. This idea popped up in 1985; it's called propagation; it started the 2nd wave of neural nets in the late 80's. Compute partial derivative with respect to each weight coordinate and update:

$$W \hookleftarrow W - \eta \nabla_w L(X^k, Y^k, w).$$

There are 2 schools: use stochastic gradient descent with 1 sample, vs. compute the entire sum for all samples. Use stochastic gradient descent because in training sets, many samples are redundant. You're wasting computation computing an accurate gradient.

The number of w's is typically in the range $[10^8, 10^9]$. The amount of computation to compute 1 output (the size of the network) is in $[10^9, 10^{11}]$. (Because of shared parameters there are fewer weights than nodes.) The number of examples is in $[10^6, 10^9]$. We have armies of people sitting around labelling images. We can also use tagged images. We want to predict hashtags

We also flip, distort the image for better training.

You can train a reasonably sized network in a week on a good laptop; in facebook/google we do it in a few hours with parallel processing.

Suppose it costs 0 to find a global minimum. Look at space of x's; there is some hidden partition. Two questions:

1. Suppose can get global minimum, how many layers do you need? 2, but the middle layer is exponentially large. (If the answer is 2 what is the question.) Given a function defined by a lot of (X, Y) pairs. Under reasonable smoothness assumptions, every function can be arbitrary approximated by sigmoids

$$f = \sum_j w_j \phi_j (\sum_k u_{ik} x_k).$$

(cf. Kolmogorov superposition theorem) People in pattern recognition, theory, have been hung up: that's the way to do things, why would you do anything else? SVM's are not efficient. Sample classification.

10^8 eqn for 10^9 param want something that's stable, works on new examples. Generalization ability. Retrain with a few more examples, the top layer, and is good for new species. Is it out of the question that things done layer by layer? If you are facebook or google, you're better off spending your money getting more samples.

Layer by layer supervised don't really work. It's not clear how to do that because the loss is only defined at the output.

It's common to add a regularizer to the loss function. It's like a prior.

$$L = \cdots + \lambda\varphi(w).$$

Think of SGD as getting a noisy estimate of the gradient. In practice we minibatch over samples of 100 because GPU's are faster when you do this.

Convergence is noisy and random. It creates a preference for solutions that are robust: relatively flat. Bouncing around increases the value of the objective. Otherwise it's smaller and will not affect the average of the loss. Stochastic methods prefer the flatter methods. The regularizer says I want the solution to be flat.

A lot is known about polynomials with Gaussian random coefficients (random matrix theory). Look at the critical points of these functions, as a function of the index of the critical point. (If there's a huge amount of saddle points with roughly equal number of dimensions going up and going down.)

We can plot a histogram of the saddle points with equal number of points going up and down. What about those with most dimensions going up? For 0 dimension going down, the loss function is clustered. Histogram of how many for each value of loss. Pretend that loss function is polynomial.

1. Regardless of optimization algorithm, you will get trapped where there are a lot of local minima. You almost always get the same value of the loss. (Never trapped in high loss.)
2. Once you minimize loss function... What you care about is average of loss function over samples you've never seen before. If you did too well on the data you have, you probably overfit. The solution with smallest loss is probably not what you want.

The game is to minimize the function on samples you've never seen before!

Dropout: in the top layers, you do as if half the nodes do not exist. It's a random subset that's different at each step. You can use only one half of the resources at each step. It improves the performance. EH: Bound on L^1 norm of w? No. $L^1 - L^2$ doesn't help. Size of weights don't matter because halfway rectification. You have the same function scaled by 2. You get some kind of generalization bound.

There are 2 challenges: what's the representational power of these systems: why is it good to stack things up. From analysis, Mark Tiger: if you make hypotheses that the signals are noise filtered, and you want to recover the filter, then stacking the layers helps.

Also recurrent networks: the matrix wraps on itself. It's a dynamical system where the state at time $t+1$ is $x_{t+1} = [f(x_t)]^+$. This goes with time.

Bring 2 vectors close if they represent the same object. "Is it Alex?" Some number in cosine space. Take the closest. Too large: threshold. It's not part of the training. Threshold built into argument. Define argument arbitrarily.

If you want to build an image search technique—give me all images that match a sentence—train an embedding. Tomasz Nikolov, word2vec. Images to vectors. Cost: all images that match query closer than images than don't match.

Google uses this technique for image search, and uses whether people click on the image to further train.

If you make them small, Because number of parameters is more than samples, each solution is degenerate; lots of weights are ignored; there are lots of directions where loss function doesn't change of is not sensitivity. This makes the function easy to optimization, find good local minimum. If you try to train a smaller network than necessary, there will be bad local minimum. If you choose the right size, it becomes hard. In regression if number of samples equals the number of dimensions you can get infinitely bad condition number.

Proper vs. improper learning. Improper easier: to learn an object of size 1000 learn it with 10000 parameters.

Add another dimensions, get around local minimum without getting trapped. The likelihood to get trapped becomes 0.

Why does this generalize? Size your network according to number of samples? SVM: overparametrize your function like crazy but regularize the hell out of it.

Oversize the neural nets like mad, and then regularize them. The fact that you can't actually get to the minimum is also a regularizer. There are many effects we don't understand.

AN: We want generative model for the data, so that if you apply your optimization method, there is an efficient procedure so that if you apply nearest neighbor, you get what you want? Embedding a vector space is a side effect. I'm interested in any loss function. I'm just interested in producing a vector which categorizes cats, dogs, etc. That's too specific. This model is ridiculous because the hypothesis is wrong. What is it as a function of the number of layers? Bounded width. Deep learning: forget about 2 layers, work with many layers.

How many layers do we need for a particular task or class of tasks?

How many layers do we need as a function of the class of the function you want to find?

Let's say I have a collection of images. I want my function invariant to shift. Is there a general statements about how many elements I need to do a good job? Assuming we can find a local minimum. Here's this class of function: with 2 layer need exponential, with $\ln n$ layer need linear or quadratic...

We can't separate 2 from 3 in circuit complexity.

Why do we always get a solution regardless of where we start from?

The geometry of the objective function. How spread in the space? Can you get from minimum by just going down, or have to search around space?

Can we come up with better algorithms? Badly behaved, conditioned things. Conditioning, spectrum of Hessian at every point well-behaved?

Conjecture: if you have a particular problem, ex. 1000000 images from ImageNet, if you train neural nets of various sizes, for any solution you find, the number of dimensions is only a function of the problem, not of the architecture: augment dimensions that are flat.

Are there situation where deep learning fails, but something else works? There are situations where you have a good understanding of the data and can get a solution directly.

Optimization with multiple layers, overfitting prevention, all mixed found. Everything found the hard way by experiments. Any theoretical framework would be good. Put sigmoid: logistic regression. Hinge: SVM. Whatever ML can do can be done with deep learning? No, find better methods (deeper learning...)

Big nuts to crack: unsupervised learning. I only have Y's, no X's. Most of our learning is unsupervised. We learn the structure of the world by observing it. We learn object permanence Thsi takes several months.

We can't do this except with text. You can predict the distribution of words that follow.

If you want to apply this to learning physics in the world, you need video. There is a lot of fuzziness. You can't predict what exactly the world will look at. How can you represent a distribution/ensemble, etc. of video frames? There's a lot of literature that doesn't work.

Represent a single frame with hidden variable theory: things that you don't observe, like person's intentions. Depending on wind, unpredictable, if you can infer hidden variabels you can make a crisp predictions.

2 Language models

For example, word2vec. These models give word embeddings that capture the meaning for the word. Given a corpus (sequence of words), map words w to vectors $v_w \in \mathbb{R}^d$ where $d = 300$ say.

This has been studied fo many decades.

A recurrent neural net is given text. Given words $w_1, \ldots, w_5$, predict the next word. The first layer maps to those vectors. There's a tradition of neural nets predicting the next word from the last 5. In what way does the embedding make sense?

1. $\langle v_w, v_{w'} \rangle$ correlates with humanly judged similarity.
2. It solves analogies. Given a testbed of 1000 analogies. To solve man:woman::king:?. Find w such that $v_{\text{man}} - v_{\text{woman}} = v_{\text{king}} - v_w$. This works $70 - 80\%$ of the time.

This is plausible neurally. Mitchell, fMRI experiments: capture brain activation in 50000 voxels. Present a person with words. Using word embeddings, given activation patterns for 50 words and then a new word, you can predict whether a voxel will be on with $> 50\%$ accuracy.

One version of word embedding:

$$\mathbb{P}(w_1, \ldots, w_5) \propto \exp\left(\left\langle v_{w_6}, \frac{1}{5}\sum_{i=1}^{5} w_i \right\rangle\right).$$

Human experiments: if you get a sentence scrambled you can probably unscramble it. Bag of words: scramble the local order.

This is a simple example of an energy-based model. What are the models actually representing?

- Understand these models (how they capture meaning).
- Understand the training algorithm.

People tried to explain these in various ways. The things to realize is that these are discriminative models. Here we want to predict the sixth word from the five words—these are called **discriminative models**.

If we want to prove any type of theorem we have to hypothesize some distribution on the input, a **generative model**.

Empirically it's been discovered that the embeddings have the property that

$$p_{ww'} = \mathbb{P}(w, w' \text{ co-occur in window of size 5}).$$

Levi-Goldberg find these models fit $\langle v_w, v_{w'}\rangle \approx \ln \frac{p_{ww'}}{p_w p_{w'}} = PMI(w, w')$. This suggests the matrix of PMIs is approximately PSD with low rank. SVD is not the best way; you need to put a weighting on it. The correct way is to put a weighting on the terms.

SVD is finding a rank d approximation, finding B of rank d minimizing $\sum |A_{ij} - B_{ij}|^2$. We need to put a weighting w_{ij}. (Weighted SVD is NP-hard for $d \ge 3$, perhaps $d \ge 2$. But in practice this works.)

Initially linguists just took $p_{ww'}$. Then they found all types of reweightings. They found the PMI worked well.

Model: there is a discourse vectors $c_t \in \mathbb{R}^d$. Each direction has some semantic meaning. Nature does a random walk for c_t on the space in $[-1, 1]^d$, and

$$\mathbb{P}(w|c_t) \propto \exp(v_w \cdot c_t)$$

(Mnih-Hinton 2007). We introduced the random walk; we integrated over the random walk to get a closed form.

The new things is that in order to compute our closed-form expression, we assume the v_w's are spatially isotropic: the global distribution is $sN(0, I)$ where s is a scalar. The v_w's are not unit vectors—they have information. Assuming $N(0, I)$ distribution allows us to calculate the messy integrals.

$$\ln p_{ww'} = \frac{\|v_w + v_{w'}\|^2}{2d} - 2\ln Z + \underbrace{\varepsilon_{n,d}}_{\to 0}.$$

We have $\ln p_w = \frac{|v_w|^2}{2d} - \ln Z + \varepsilon'_{n,d}$.

We can prove the analogy solving more rigorously. Why is queen the right answer? For most words χ,

$$\frac{\mathbb{P}(\chi|\text{king})}{\mathbb{P}(\chi|\text{queen})} \approx \frac{\mathbb{P}(\chi|\text{man})}{\mathbb{P}(\chi|\text{woman})},$$

ex. eat, walk. But for x =dress, John, Elizabeth this is not true. So we minimize

$$\min_{\omega} \sum_{\chi} \left| \ln \frac{\mathbb{P}(\chi|\text{king})}{\mathbb{P}(x|w)} - \ln \frac{\mathbb{P}(\chi|\text{man})}{\mathbb{P}(\chi|\text{woman})} \right|.$$

Polysemy problem: Words like "tie" have many meanings. These embeddings represent words by a single vectors. What does it do for polysemous words. We found a survey by linguists a few years ago. "Obviously, you cannot just use word occurence counts for polysemy."

We set up sparse coding on $\{v_w\}$ with basis size ≈ 2000 with sparsity 5, such that every word is a linear combination of 5 words. The five nonzero coefficients capture the meanings.

We created artificial polysemous words: take random words w, w' and create w_{new} the same token representing these words. This sequence of letters represents 3 different words. We find that $v_{w_{\text{new}}} = v_w + 0.2v_{w'}$, say.

What does the basis correspond to? Atoms of discourse. Here are 2000 directions corresponding to things people talk about (social media, cell biology, Indian cities, etc.).

Look at tie: get atoms corresponding to clothing, rope, games.

What about multiple languages?

3 takeaways:

1. Current efforts to understand word2vec.
2. All kinds of linear algebra that are NP-hard so you will never learn in linear algebra calss.
3. Fitting a models are like fitting, factoring matrices. The more general problem is for tensors (Rong Ge).

3 Simple, efficient, and neural algorithms for sparse coding

We'll think about algorithms for sparse coding by connecting them to approximate gradient descent. How iterative algorithms on nonconvex algorithms make progress.

Olshausen and Field introduced sparse coding to try to understand a scientific mystery: the response properties of neurons—they were trying to solve an optimization problem.

They took many natural images, broke them into patches, and applied sparse coding: find a basis for them. They look like Gabor filters: localized, bandpassed, and oriented.

Contrast to a theorist's swiss army knife of singular value decomposition. You'd find a noisy, different basis that is difficult to interpret.

Applying sparse coding gives much more meaningful bases.

Are there efficient, neural algorithms for sparse coding with provable guarantees? The algorithms should not just be polynomial time, but made of simple building blocks so they might be implemented neurally.

3.1 Olshausen-Field Update Rule

Many types of data are sparse in an appropriately chosen basis: images, signals, etc. Think of them as n vectors; we're given p of them. We have a $n \times m$ dictionary A,

$$Ax^{(i)} \approx b^{(i)}.$$

We want the representation to be simple. Sparse recovery: if you were given A and b can you find a sparse x such that $Ax \approx b$. But now the meat of the issue is to learn A from examples! Given only the b's, you're promised that they come from A and k-sparse x's. You could add a penalty function that become denser and denser; you can use analytic sparseness, etc.

For example, 1-sparse reduces to k-means: look for m means.

Here's the usual nonconvex version:

$$\min_{A,x^{(i)}} \sum_{i=1}^{p} \left\| b^{(i)} - Ax^{(i)} \right\| + \sum_{i=1}^{p} L(x^{(i)})$$

where the last term is a nonlinear penalty function to encourage sparsity. (We can do a hard constraint like ∞ for $k+1$ nonzeros, or something softer.)

We'll give iterative rules and talk about why they work.

OF's approach is as follows.

- Architecture: There is a 3-layer network. Given an image patch, keep track of the residual, how much of the image you haven't represented yet. The dictionary is stored in the strength of connections between wires. Start with image, add to residuals, multiply by A. There is feedback to penalize dense x.

  The network performs gradient descent on $\|b - Ax\|^2 + L(x)$. Alternate between

  1. $r \hookleftarrow b - Ax$
  2. $x \hookleftarrow x + \eta(A^T r - \nabla L(x))$.

A is updated through Hebbian rules (strengthen/weaken connections). There are no provable guarantees but this works well. It works well empirically: given natural images, it makes Gabor filters that are interesting. You're happy by the result because it has low reconstruction error with sparse entries.

The generative model we assume will have sparse basis. Do we find that basis?

The iterative rule finds a basis. In the model there is a global minimum.

Other approaches and applications.

- signal processing/statistics: MOD, kSVD
- Machine learning

  Generative model:

1. given unknown dictionary A,
2. generate x with support size k u.a.r.; choose non-zero values independently, observe $b = Ax$.

We can also add in noise, $Ax + \beta$. Many extensions continue to work, but we'll focus on the simple model.

Spielman, Wang, and Wright (13): works for full column rank up to sparsity $\sqrt{n}$ ($m \leq n$). If makes more sense for A to have many more columns to rows—it's much more flexible. Often you mix and match dictionaries—sparse in some combination of basis elements.

Arora, Ge, and Moitra (14): works for overcomplete, μ-incoherent A up to sparsity $n^{\frac{1}{2}-\varepsilon}/\mu$.

Agarwal et l. (14): overcomplete, μ-incoherent A up to $n^{\frac{1}{4}}/\mu$ via alternating minimization. Use L^1 optimization as subroutine.

Barak, Kelner, Steurer (14): works for overcomplete A up to $n^{1-\varepsilon}$, running time exponential in accuracy.

Arora, Ge, Ma, Moitra 14: There is a variant of OF-update rule that converges to the true dictionary.

New update rule:

1. $\widehat{x}^{(i)} = \mathit{threshold}(\widehat{A}^T b^{(i)})$: how correlated your vector is with basis elements so far, threshold to zero out nonzero entries. Take that as representaiton
2. $\widehat{A} \hookleftarrow \widehat{A} + \eta \sum_{i=1}^{q} (b^{(i)} - \widehat{A}\widehat{x}^{(i)}) \operatorname{sign}(\widehat{x}^{(i)})^T$. Update with rank-1 outerproduct: residual error.

Nice:

- The samples arrive online. In contrast, previous provble algorithms might need to compute a new estimate from scratch, when new samples arrive.
- The computation is local: a neuron looks at neighbors and do thresholds.
- The update rules is explicitly Hebbian: neurons that fire together wire together.

The weight $\widehat{A_{i,j}}$ is the product of the activations at the residual layer and the decoding layer.

What is neurally plausible? Skip.

Approximate gradient descent: we give a general framework for designing and analyzing iterative algorithms for sparse coding. Usual approach is to think of minimize non-convex function

$$\min_{\widehat{A},\text{coln-sparse}\widehat{X}} E(\widehat{A}, \widehat{X}) = \left\| B - \widehat{A}\widehat{X} \right\|_2^2$$

Why should a iterative problem make progress on nonconvex function? What if think of it as minimizing unknown convex function.

Think of X as unknown:

$$\min_{\widehat{A}} E(\widehat{A}, X) = \left\| B - \widehat{A}X \right\|_F^2 .$$

Now the function is strongly convex, and has global optimum that can be reached by gradient descent.

Separately convex in each variable.

New goal: prove that with high probability the step (2) approaches the gradient of this function.

Just have some nontrivial inner product to not get stuck. This is where we use distributional properties.

Conditions for convergence (convex optimization): consider the following general setup:

- optimal solution z^*

- update $z^{s+1} = z^s - \eta g^s$.

This works even if it's correlated with the gradient:

Definition 3.1: g^s is $(\alpha, \beta, \varepsilon_s)$-correlated with z^* if for all s,

$$\langle g^s, z^s - z^* \rangle \ge \alpha \|z^s - z^*\|^2 + \beta \|g^s\|^2 - \varepsilon_s.$$

Theorem 3.2: If g^s is $(\alpha, \beta, \varepsilon_s)$-correlated with z, then

$$\|z^s - z^*\|^2 \le (1 - 2\alpha\eta)^2 \left\|z^0 - z*\right\|^2 + \cdots .$$

Take usual proof almost verbatim; can make progress as long as correlated. Fresh randomness golf you out of where you are.

1. $\widehat{x}^{(i)}$ =threadhold($\widehat{A}^T b^{(i)}$).

   Formualte decoding lemma.

2. Update $\widehat{A}$.

   Calculate expectation of column-wise update rule. Expectation is $A_j - \widehat{A_j}$. The rest is systemic bias $\xi \mathbb{E}_R[\widehat{A}_R \widehat{A}_R^T] A_j$. Then auxiliary lemma.

   Various conditioning trips.

An initialization procedure: we give an initialization algorithm that outputs $\widehat{A}$ that is column-wise δ-close to A for $\delta \le \frac{1}{\text{poly}\log(n)}$, $\left\|\widehat{A} - A\right\| \le 2$.

1. Choose samples b, b'. There's a reasonable chance they intersect in 1 element (constant/polylog chance).

2. Calculate $M_{b,b'}$. Filter the guys which share that column.

3. Hope there is one large eigenvalue and others noticably smaller.

This happens when $\text{Supp}(x) \cap \text{Supp}(x')$ is a singleton.

Further results: adjusting an iterative algorithm can have subtle effects on its behavior. Think accelerated gradient method vs. high ball method (?).

We can use our framework to systematically design and analyze new update rules. "Crosstalk" between columns. Once know where error comes in, can remove systemic bias, by carefully pojecting out along direction being updated.

1. $\widehat{x}_j^{(i)} = \textit{threshold}(\widehat{C}_j^T b^{(i)})$ where $\widehat{C}_j$ has all columns except $\widehat{A}_j$ projected to the orthogonal complement of $\widehat{A}_j$.

2. Update $\widehat{A}_j$.

Think of the convex programming problem you wish you had.

Aren't you hiding in original initialization? All these things need careful initialization.

It's easy to analyze when you're very close to the minimum. Here is the medium range $\frac{1}{\text{poly}\log(n)}$.

Summary

1. Online, local, Hebbian algorithms for sparse coding that find globally optimal solution (whp).
2. Introduced framework for analyzing iterative algorithms by thinking of them as trying to minimize unknown convex function
3. The key is working with a generative model.
4. Is computational intractability really a barrier to a rigorous theory of neural computation?

Conditional pairwise independent.

Put other alternate minimization into this framework, and engineer more alternate minimization.

4 Tensor decomposition

Two parts:

1. Many unsupervised learning problems can be rephrased as tensor decomposition problems.
2. Stochastic gradient descent for tensor decomposition.

One problem is the mixture of Gaussians problem. Observe samples from Gaussian distributions. Assume they have the same covariance matrix and are spherical. For each sample, you don't know which Gaussian it came from. Try to find the center of the Gaussian distributions. The k components are the k centers.

Many of the problems discussed today, like sparse coding, falls in this category. Find vectors that can sparsely represent all the given vectors.

All these problems are necessarily nonconvex because the solution we are looking for are components $u_1, u_2, \ldots, u_k \in \mathbb{R}^d$. Suppose the objective function is $L(u_1, u_2, \ldots, u_k)$. The components do not come in specific order: if we swap then it's still the same solution. If we take a convex combination then it's no longer an optimal solution. Because of the symmetry there are many equivalent global optimum solutions.

Problem 4.1 (Independent component analysis)**:** There are n unknown independent sources (people talking) and n signals observed (microphones). What are the original sources?

Suppose x has independent components $\mathbb{E}x = 0$, $\mathbb{E}xx' = 1$. (Ex. Consider $x \sim_R \{\pm 1\}^n$. The general notion is that there is a measure of non-Gaussianity (if all components are Gaussian it looks the same from every direction))

There is an unkown linear transform $A \in \mathbb{R}^{m\times n}$.
Observe $y = Ax$.
Goal: find (approximate) A.

One way to solve this problem is to use tensor decomposition.

Tensors are higher-dimensional arrays. We consider 4-dimensional tensors $T \in \mathbb{R}^{n^4}$. A tensor is rank 1 if it can be written as a product of tensors, $T = x \otimes x \otimes x \otimes x$ (we consider symmetric tensors). Then $T_{i,j,k,l} = x_i x_j x_k x_l$. A low rank decomposition is $T = \sum_{i=1}^{4} x_i x_i^{\otimes 4}$.

Define

$$T(u) = \sum_{i,j,k,l} T_{ijkl} u_i u_j u_k u_l.$$

This is a degree 4 polynomial. For this particular tensor, $T(u) = \sum_{i=1}^{r} \lambda_i \langle x_i, u\rangle^4$.

This is similar to spectral decomposition: breaking up into a sum of r different rank 1 components. The benefit from tensor decomposition is that it is unique.

How do we construct tensors? Use the 4th order cumulant $\kappa_4(X)$. We need the following properties:

1. If X, Y are independent, $\kappa_4(X+Y) = \kappa_4(X) + \kappa_4(Y)$.
2. (looks like degree 4 homogeneous polynomial) For $C \in \mathbb{R}$, $\kappa_4(CX) = C^4\kappa_4(X)$. ($\kappa_4(X) = \mathbb{E}[X^4] - 3\mathbb{E}[X^2]^2$ when $\mathbb{E}[X] = 0$.) For Gaussian this is 0. If it's nonzero it's far from Gaussian.

Using this cumulant you can construct a tensor with low-rank form.

Claim 4.2: If $T(u) = \kappa_4(u^T y)$ then $T = \sum_{i=1}^{n} \kappa_4(x_i) A_i^{\otimes 4}$.

Proof.

$$\begin{aligned}\kappa_4(u^T y) &= \kappa_4(u^T Ax)\\ &= \kappa_4\left(\sum_{i=1}^{n} \langle u, A\rangle x_i\right)\\ &= \sum_{i=1}^{n} \kappa_4(x_i)\langle u, A\rangle^4 .\end{aligned}$$

□

People have tried different functions. Some work better in practice. In theory this is easiest to analyze because it's a degree 4 polynomial.

AN: If look at characteristic functions, product of functions of 1 variable. Moment generating: anything takes sum into product.

$$\mathbb{E}e^{iu^T y}$$

This breaks down as a product. Functions of 1 variable? (Santosh? ICA paper.) It's important to represent function in finite way. When something breaks down into product, it's a different way of representing it simply.

In general tensor decomposition is NP-hard. Our problem we can solve in polynomial time; there are many algorithms. WLOG assume A is orthonormal ($A^T A = I$). We have $\mathbb{E}[yy^T] = \mathbb{E}[Axx^T A^T] = AA^T$. We compute a whitening matrix W such that $W^T AA^T W = I$. (Use matrix factorization.)

We can change the problem to $Z = W^T y = W^T Ax$. WLOG can always assume linear transformation orthonormal.

The algorithm works like this.

1. Pick random vectors $a \in \mathbb{R}^n$. Compute $T(a, a) = \sum_{i=1}^n \kappa_4(x_i) \langle A_i, a\rangle^2 A_i A_i^T = ADA^T$. The entries of D are likely to be distinct. Just do SVD of $T(a, a)$ to find A.

Do perturbation analysis: even if don't know exactly, we can approximate A.

In practice the tensor is not exactly row rank. This algorithm is not so robust. Then I show how we can use basic optimization techniques to solve this kind of tensor decomposition problem.

Tensor decomposition can solve mixture of Gaussian (with arbitrary covariance), hidden Markov models, topic models, stochastic block models, models for trees (phylogeny reconstruction).

Let $f(u) = -\frac{1}{2}T(u) = \sum_{i=1}^n \langle u, A_i\rangle^4 = \left\|A^T u\right\|_4^4$. We want

$$\max_{\|u\|_2=1} \left\|A^T u\right\|_4^4 .$$

Rotating to basis A, the L^2 norm does not change.

$$\max_{\|u\|_2=1} \|u\|_4^4 .$$

This is maximized when u is one of the basis vectors. If we can find a local max, then we can find one component of the tensor. This is still nonconvex; there are many saddle points.

The above is just a motivating example. It doesn't have any bad local maximum. They are global maximum and correspond to meaningful components. If you start from bad points the gradient will not move. It's not clear why you can use gradient descent. People use second-order optimization techniques: Find a positive eigenvalue in your Hessian.

Guaranteed to find one of local solutions. We come up with the definition of strict saddle functions.

Definition 4.3: Critical point of f is x such that $\nabla f(x) = 0$. If x is neither a local max nor min, then we say x is a saddle point.

Assume $f \in C^\infty$. $f(x)$ is a strict saddle if for any point x one of the following holds.

1. x is close to a local minimum. (Assume f is strongly convex there so the Hessian is strictly convex.)
2. $\|\nabla f(x)\|_2$ is large
3. $\lambda_{\min}(\nabla^2 f(x)) < -\gamma$. It has a negative direction, and is upperbounded by $-\gamma$.

Then you can use a second-order algorithm.

(On a manifold redefine the gradient to be the projection onto the tangent space. Hessian with Lagrange multiplier.)

In 2 and 3 you are guaranteed to make fixed progress, so that you can get close; then you are in case 1.

We show not only you can optimize it, you can optimize this using stochastic gradient descent. Local minimizations are permutations of global minimizers, tensor decomposition.

What do I mean by a stochastic gradient algorithm?

$$x^{(t+1)} \hookleftarrow x^{(t)} - \eta(\nabla f(x^{(t)}) + \varepsilon_d)$$

where the ε_d is a random variable that capturres the error because we are not using all samples. We have

$$\forall x, \mathbb{E}[\varepsilon_t | x^{(t)}] = 0,$$

i.e., we have an unbiased estimator for the gradient.

Theorem 4.4 (GHuangJinYang, COLT)**:** If $f(x)$ is a strict saddle then stochastic gradient converges to a local minimum in polynomial number of steps.

This is different from Ankur's talk: no matter where you start, as long as your function has this property, your function will always converge.

Ankur broke the symmetry by the initialization algorithm (unique permutation of components closest to solution). Here we don't have that, but because we are doing SGD, can break symmetry so will converge to one of multiple equivalent solutions.

The intuition is:

1. Trying to optimize $f(x)$, a strongly convex function.

2. As long as your step size is not too large, we will in expectation make progress after 1 step.

3.

What if we start close to saddle point? Consider $x^2 - y^2$. Then $\nabla f = (2x, -2y)$. If ε_d is a standard Gaussian, or any other whose covariance matrix is something we have a sense of (ex. I), then we can recursively compute $\mathbb{E}[(x^{(t+1)})^2] = (1-2\eta)^2\mathbb{E}[(x^m)^2] + \eta^2$. We can solve this recursion. In a small number of steps the expectation converges to $\theta(\eta)$. On the other hand, for y you get $(1+2\eta)$ and this goes to infinity. It becomes large after $O\left(\frac{1}{\eta}\right)$ steps. In $\frac{1}{\eta}$ steps, the expectation of y^2 is larger in the x direction. The same intuition holds in more complicated settings: you might have more directions that have positive eigenvalues, but they will be bounded in $\mathbb{E}\bullet^2$, those with negative eigenvalues will $\to -\infty$. This is cheating because for this function the Hessian is constant. The Hessian is the same no matter how far you go. In the real proof what's hard is to show you don't go very far so we can use smoothness of the Hessian to make sure that even if it's not constant the argument goes through.

Finally, the construction for tensor decomposition is as follows. Let $T = \sum_{i=1}^{n} A_i^{\otimes 4}$. We happen to have a construction ($u_i \in \mathbb{R}^n$, $\|u_i\| = 1$)

$$\begin{aligned} f(u_1, \dots, u_n) &= \sum_{i \neq j, 1 \leq i \leq n} T(u_i, u_i, u_j, u_j) \\ &= \sum_{k=1}^{n} \langle A_k, u_i \rangle^2 \langle A_k, u_j \rangle^2 \end{aligned}$$

Compute just from T, u_i, u_j. Lemma: Local min of f will have $u_i = \omega A_{\pi(i)}$.

Write out the u_i in the basis of A_i's.

Zero when disjoint support. Otherwise positive contribution: nothing will cancel it. One term is 0 only when u_i, u_j have disjoint support when represented in basis of A. n vectors pairwise disjoint support. None empty.