Deep Learning Introduction

This material is intended as a brief overview of a very large field, with the limited goals of making connections with earlier material in the course, defining some terms, and outlining some open issues.

The general motivating problem is to classify high-dimensional quantities – for example, to determine whether photographs contain a cat, whether an audio stream is spoken in English, or whether some text is relevant to a search query. Since our brains can do these tasks, it is hypothesized that a neurally-inspired architecture may be a good way to solve these problems.

Architectures

One can also view deep learning architectures as natural outgrowths of simpler classifiers. The simplest is a linear discriminant – identification of a set of weights w_i, that enable observation vectors x_i to be classified into two categories, based on the value of $y = \sum_i w_i x_i$. The Fisher linear discriminant is of this form. The weights are determined by solving an eigenvalue problem, which maximizes the ratio of the between-class variance to the within-class variance. Support vector machines also yield a decision variable in this way, but the quantity that is extremized to find the weights is different: the number of classification errors is minimized. This architecture is also that of the "perceptron": a linear weighting of binary inputs, followed by a decision threshold.

These classification methods only work if the observations in the two classes can be separated by a hyperplane. When this is not the case, it is natural to seek a solution by embedding the original observations in a higher-dimensional space, anticipating that in the larger space, the points can be separated by a hyperplane. For example, one could create new coordinates from the raw observations x_i by also considering their squares and pairwise products. This leads to a classifier of the form $y = \sum_i w_i x_i + \sum_{i \le j} q_{i,j} x_i x_j$ -- a quadratic discriminant. The new (and larger) set of weights can be found by the same approach that worked for linear classifiers, but now applied to the augmented set of variables. If, for example, one category consists of points near the origin and the other category consists of points that are far from the origin, then no hyperplane will separate these two clouds, but a quadratic discriminant -- $y = \sum_i x_i^2$ -- will do a great job.

The derived variables (the nonlinear functions of the original variables) the can be regarded as "features", which, hopefully, are more useful than the original variables for classification.

The features needn't be limited to polynomials or pairwise combinations of the original variables. So it is a real challenge to decide how to choose them. It is natural to look to biology for inspiration, since our brains appear to be able to solve classification problems quite well.

A useful heuristic is that the features (i.e., the nonlinear combinations of the input variables) should be constructed in a local fashion. That is, if the input consists of images, the features might be constructed from pixel values at nearby points; if the input consists of sound, the

features might be constructed from amplitudes at nearby times. We might consider these adjacent signals to be inputs to a feature-extracting "neuron", which then sums them, according to some set of weights, and then applies a nonlinearity. Thus, at each location or at each time, we would create a feature y_j via $y_j = F_j(\sum_{i \text{ is near } j} w_{i,j} x_i)$.

If this works – that is, that we can now create a linear classifier from the y_j's – we've now built an artificial neural network with one hidden layer. There's the input layer consisting of the inputs, the "hidden" layer whose outputs are the features y_j, and a final layer that computes a weighted sum of the y_j's to generate a decision variable.

But if this fails, there are evident ways that we could extend the idea. In particular, loosely inspired by a caricature of cortical visual processing, we could create a second level of features z_j from the y_j's, and try again to find a linear classifier. We could continue to add more and more layers, with each layer creating features by additively combining local signals from the previous layer, and applying a nonlinearity. The features become more and more abstract. And even if each layer has only local connectivity, eventually, features at a sufficiently deep layer will have access to the entire input. This is the "deep neural network" architecture.

Determining network parameters

Once the architecture has been chosen, the challenge of determining the parameters – the weights, and the nonlinear functions -- looms. As formulated above, the number of model parameters is extraordinarily large – in each layer of an image-processing network, there is a weight $w_{i,j}$ for each location j in the image, and, for each location i in its neighborhood. Additionally, the "activation functions" F need to be chosen and parameterized. So without further restrictions, there could be thousands to millions of parameters to fit.

Reducing the parameter count

We can dramatically reduce the parameter count by adding an assumption: that the features to be extracted are independent of location. That is, the weights $w_{i,j}$ depend only on the relative locations in the current layer (j) and the input from the previous layer (i): $y_j = F_j(\sum_{i-j \in L} w_{i-j} x_i)$.

For image processing, we'd interpret the subscripts as 2-vectors, indicating horizontal and vertical location in the image. Similarly, for movies, the subscripts would be 3-vectors, two coordinates for position and one for time.

With this constraint, the signals created by weighted sums are a convolution of the previous layer's outputs.

We could further require that the activation functions F_j are identical too. This assumption of location-invariance can be applied to individual layers, or to all layers – the latter, resulting in a "convolutional neural network."

We can retain the spirit and potential advantages of convolution without requiring that each layer has the same number of nodes as the previous layer. A layer can downsample a previous layer, but it can also upsample – for example, by extracting multiple features at each point: $y_j^{[k]} = F_j(\sum_{i-j\in L} w_{i-j}^{[k]} x_i)$. Inspiration for this again comes from biological vision: at each retinal location, there are "ON" cells and "OFF" cells, and at each location in the visual field there are cortical neurons that respond best to bars at the full range of orientations.

Activation functions

As above, an activation function determines how a node responds to the weighted combination of its inputs: $y = F(\sum_i w_i x_i)$. Note that in the absence of an activation function, all of the layers would collapse to a single linear transformation. So they are crucial.

There are many reasonable choices for activation functions, each with their own rationales:

- RELU (rectified linear): $F(x;a) = \max(x-a, 0)$ -- a caricature of a neuron
- Quadratic: $F(x;a) = (x-a)^2$ -- corresponds to a squared distance; allows for multiplication $\frac{1}{4}(u+v)^2 - \frac{1}{4}(u-v)^2 = uv$
- Half-quadratic: $F(x;a) = \left(\max(x-a, 0)\right)^2$ – combines the ideas behind RELU and quadratic
- Exponential: $F(x) = e^x$ – analytic, works well with general linear models, no parameters
- Hyperbolic tangent: $F(x;a,\beta) = \tanh\left(\beta(x-a)\right)$ – analytic, symmetric, saturates in both directions
- Sigmoid: $F(x;a,\beta) = \frac{e^{\beta(x-a)}}{1+e^{\beta(x-a)}}$: also saturates in both directions, but asymmetric. Mimics channel-opening?

All of the above can be scaled by a multiplier, but this is redundant with changing the weight of its output at the next layer.

We also need a special kind of activation function at the output layer, to render a classification. These functions map a set of inputs x_i at the next-to-last layer to a set of final outputs p_i, each representing the likelihood of membership in each of the possible categories. The usual choice is the "softmax", with parameter β: $p_i = \frac{e^{\beta x_i}}{\sum_j e^{\beta x_j}}$. Note that $\sum_j p_j = 1$ and each $p_j \geq 0$, so can be considered the probability that the correct answer is class i. Note also that as $\beta \to \infty$, the softmax becomes "hard", i.e., it assigns 1 to the category i with the largest x_i, and 0 to the others.

Backpropagation

Even with the parameter count reduced by a convolutional architecture in one or more layers, the number of parameters remains quite large (typically at least hundreds). Moreover, analytic solutions, such as those that yielded the linear discriminant, are not available. So, iterative procedures – "training" or "learning" -- must be used, even when there is only one hidden layer.

The identification of a basic training technique, "backpropagation" (Rumelhart, Hinton & Williams (1986) – though there were many earlier foreshadowings of this) was a major advance that enabled experimental exploration of the capabilities of artificial neural networks.

The basic idea is to define a "loss function" that is to be minimized by adjusting the weights. A simple loss function is the frequency of misclassifications; more sophisticated cost functions might take into account the severity of the errors. After initialization of the network with some (typically random) choice of weights, the loss function is computed by examining the network's outputs for collection of inputs for which the correct answer is known. The weights are then adjusted by gradient descent. That is, the derivative of the loss function is calculated with respect to each weight, and then the weights are adjusted a small amount to reduce the loss. The term "backpropagation" is used for this since the network is a nested series of nonlinearities, and application of the chain rule amounts to propagation of changes in the output back to earlier layers.

Backpropagation algorithms themselves have many parameters and options – for example, how much to change the weights at each step, and how many examples to examine in a "batch" before modifying the weights.

Room for progress

Along with recognizing the dramatic successes of deep learning in some domains (facial recognition, object recognition), it is worthwhile to be aware of the limitations and areas in which substantial improvement is likely still possible. In broad strokes, these successes represent empirical successes built on brute force, rather than on principle – and designing a deep network to solve a new task has been likened to building a bridge by reference to other bridges that have worked, but without a knowledge of physics.

Even with a reduction in parameter count via a convolutional architecture, and a well-tuned backpropagation algorithm, the size of the training set needed to reach adequate performance can be immense. Even with a large training set, it is typically the case that a deep learning network that solves one problem fails to solve related ones (e.g., it can determine whether images of isolated objects are images of a cat, but it can't do this if the image has a cluttered background.) The generalization failure is distinct from overfitting *per se* -- standard training procedures protect against overfitting by holding out a portion of the training set for validation. Proper behavior on a held-out subset of the training set is not the same as proper behavior in a new context.

One approach to the problem of generalization is to recognize that some appropriate preprocessing might be able to remove "nuisance" variables introduced by context. A simple example is that, instead of processing the light intensity values in an image, one could remove the mean level and process local contrasts. More abstractly, one would want to identify a smaller

set of features, from which one could reproduce, or at least approximate, the inputs, in a way that is sufficiently informative as to allow for the desired decisions.

This is the idea behind an “autoencoder” – an artificial neural network whose task is to create a simpler representation of the input set, for further processing a second neural network. That is, for the autoencoder, the loss function is a measure of how well the original input is reproduced.

Principal components analysis can be viewed as a basic kind of autoencoder. Each component is represented by a node, and the weighing for each node (non-convolutional) is the linear transformation that projects the data onto that principal component. If all components are represented by nodes, the input can be fully reconstituted; if one limits the nodes to a smaller number of components, one has achieved a reduction in dimension.

The autoencoder generalizes this idea to nodes with nonlinear activation functions. Importantly, simplification of the representation is usefully enforced by means other than reducing the number of nodes. A typical approach is to allow more nodes in intermediate nodes or in the output, compared to the input, but to enforce simplicity by including a term in the loss function favors output patterns in which only a few nodes are active – a “sparsity” term. “Sparse dictionary learning” is an example of this: as with a generic autoencoder, the autoencoder’s output is a sparse set of signals that is derived from the input via weights and nonlinear activation functions, but the input can be reconstructed by a linear combination of the outputs – the “dictionary”.

Differences between DL architectures and human processing

While artificial neural networks are explicitly inspired by biological neural networks, there are many ways in which they differ – both in terms of architecture and function.

Regarding architecture: real neural networks have massive feedback between layers; their function relies on dynamics, especially the interplay between within-layer processing and between-layer processing; real neurons use spikes; real networks make use of active sensation (e.g., moving the sensor); real networks make use of attention; real networks are not convolutional – they are not spatially homogeneous (e.g., we have a fovea).

Regarding function: we learn from a small number of trials; we easily generalize; errors usually result in a mis-classification into a related category rather than an unrelated one; small changes in the input (e.g., a single pixel) result in small changes in the classification.