ECBM E6040 Neural Networks and Deep Learning

Lecture #6: Feedforward Deep Networks

Aurel A. Lazar

Columbia University
Department of Electrical Engineering

February 23, 2016

Outline of Part I

2. Summary of the Previous Lecture
   - Topics Covered
   - Learning Objectives

Outline of Part II

3. Multilayer Perceptrons from the 1980s
4. Estimating Conditional Statistics
5. Parametrizing a Learned Predictor
   - Family of Input-Output Functions
6. Computational Graphs and Back-Propagation
   - Chain Rule
   - Back-Propagation in an MLP
   - Back-Propagation in a General Computational Graph

Part I

Review of Previous Lecture

Topics Covered

- Bayesian Statistics
- Supervised Learning Algorithms
- Unsupervised Learning Algorithms
- The Curse of Dimensionality

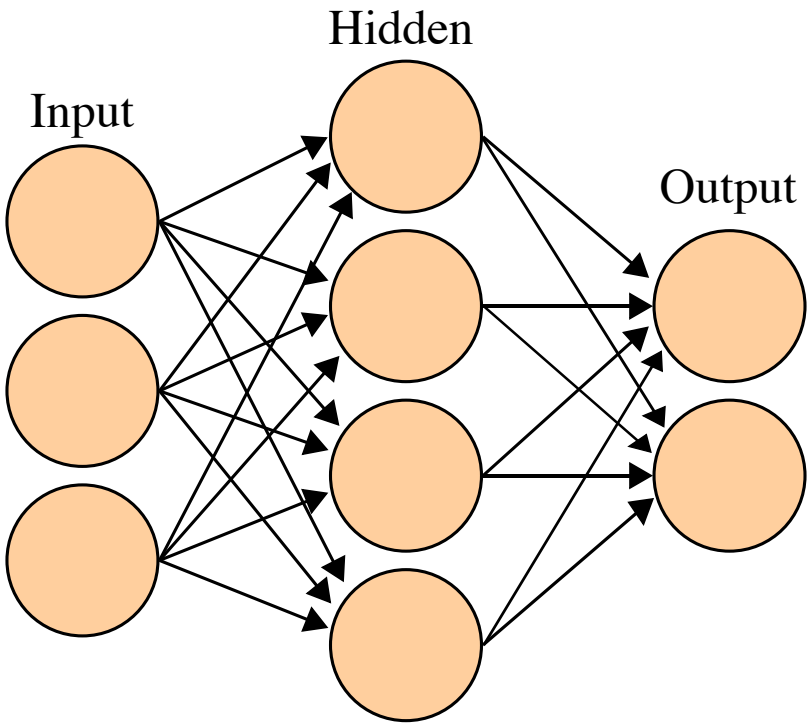
Learning Objectives

- Understanding the principles behind the frequentist and Bayesian modeling approaches to statistics.
- Learning a data representation, that preserves as much information as possible under some penalty or constraint, and that is aimed at keeping the representation simpler or more accessible than the data itself.
- Deep learning algorithms consist of: (i) a dataset, (i) a cost function, (iii) an optimization procedure and, (iv) a model.
- Curse of dimensionality: the number of possible distinct configurations of the variables of interest increases exponentially with the dimensionality.

Part II

Today's Lecture

Feedforward Deep Networks or Multilayer Perceptrons

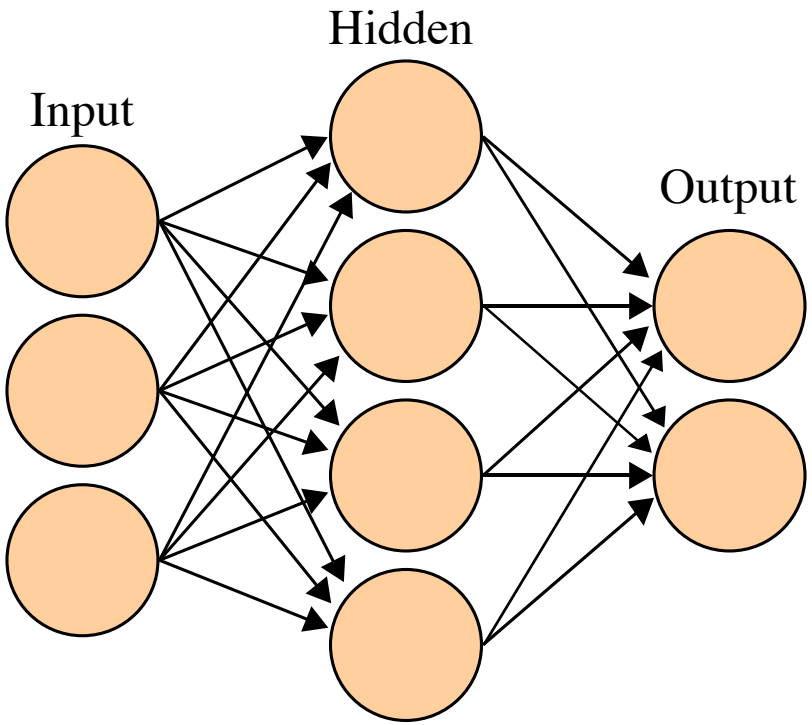


Multilayer Perceptrons (MLPs) are multi-input and multi-output parametric functions defined by composing together many parametric functions.

Each sub-function is known as a layer of the network, and each scalar layer output of one of these functions as a unit or feature.

The number of units in each layer is called the width of a machine learning model, and the number of layers is called the depth.

Feedforward Deep Networks or Multilayer Perceptrons



$\boldsymbol{x} \in \mathbb{R}^{n_i}$ $\mathbf{h} \in \mathbb{R}^{n_h}$ $\hat{\mathbf{y}} \in \mathbb{R}^{n_o}$

$\mathbf{W} \in \mathbb{R}^{n_h \times n_i}$ $\mathbf{V} \in \mathbb{R}^{n_o \times n_h}$

The hidden unit vector is $\mathbf{h}$

$$\mathbf{h} = \text{sigmoid}(\mathbf{c} + \mathbf{W}\boldsymbol{x})$$

with weight matrix $\mathbf{W}$ and offset vector $\mathbf{c} \in \mathbb{R}^{n_h}$.
The output vector is obtained via the learned affine transformation

$$\hat{\mathbf{y}} = \mathbf{b} + \mathbf{V}\mathbf{h},$$

with weight matrix $\mathbf{V}$ and output offset vector $\mathbf{b} \in \mathbb{R}^{n_0}$.

Shallow Multi-Layer Neural Network for Regression

We consider the family of input-output functions

$$\mathbf{f}(\boldsymbol{x};\boldsymbol{\theta}) = \mathbf{b} + \mathbf{V}\text{sigmoid}(\mathbf{c} + \mathbf{W}\boldsymbol{x}),$$

where $\text{sigmoid}(a) = 1/(1+e^{-a})$ is applied element-wise. The input vector is $\boldsymbol{x} \in \mathbb{R}^{n_i}$, and the output vector is $\mathbf{f} \in \mathbb{R}^{n_o}$. The parameters are the flattened vectorized version of the tuple

$$\boldsymbol{\theta} = (\mathbf{b}, \mathbf{c}, \mathbf{V}, \mathbf{W}).$$

For the loss function

$$L(\hat{\mathbf{y}}, \mathbf{y}) = ||\hat{\mathbf{y}} - \mathbf{y}||^2$$

one can show (using calculus of variations) that $\hat{\mathbf{y}} = \mathbb{E}[\mathbf{y}|\boldsymbol{x}]$.

Multi-Layer Neural Network for Regression (cont'd)

Define the weights $\boldsymbol{\omega}$ as the concatenated elements of matrices $\mathbf{W}$ and $\mathbf{V}$ and $||\boldsymbol{\omega}||^2 = \sum_{i,j} W_{ij}^2 + \sum_{ki} V_{ki}^2$. The regularized cost function typically considered

$$J(\boldsymbol{\theta}) = \lambda||\boldsymbol{\omega}||^2 + \frac{1}{n}\sum_{t=1}^{n} ||\mathbf{y}^t - (\mathbf{b} + \mathbf{V}\text{sigmoid}(\mathbf{c} + \mathbf{W}\boldsymbol{x}^t))||^2,$$

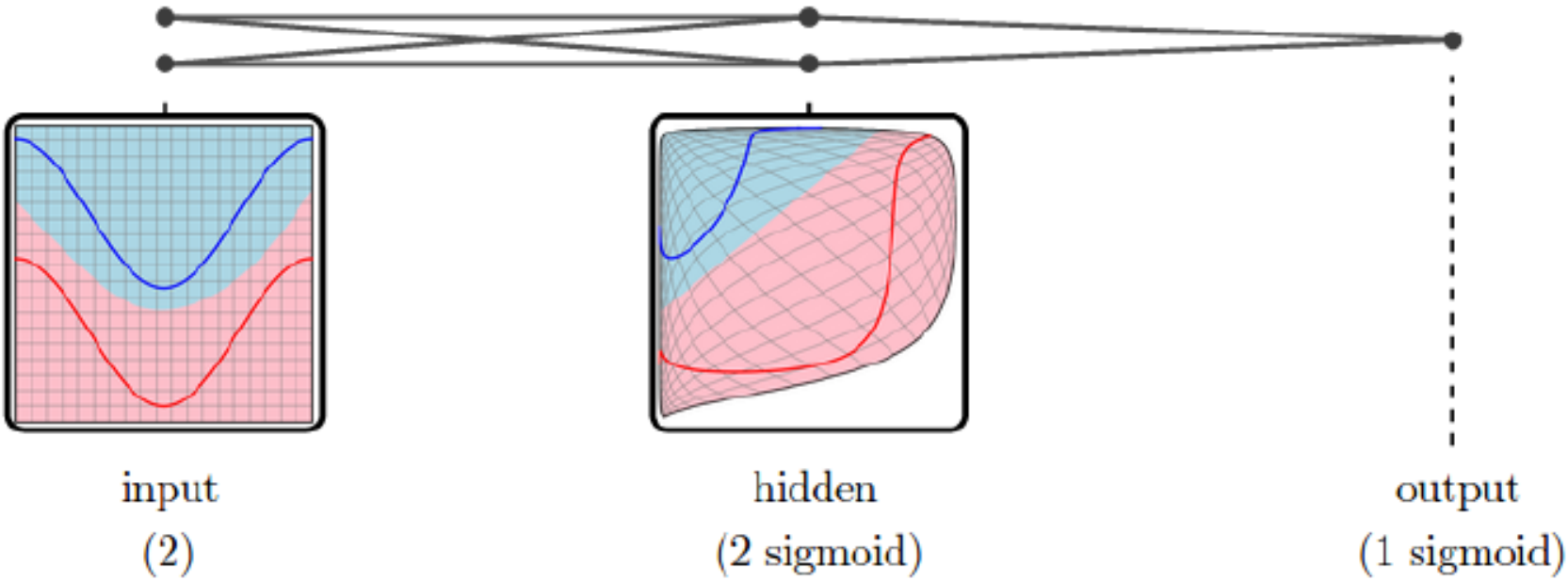
where $(\boldsymbol{x}^t, \mathbf{y}^t)$ is the t-th training (input, target) pair.
The classical training procedure iteratively updates $\boldsymbol{\theta}$ as in

$$\begin{aligned}\boldsymbol{\omega} &\leftarrow \boldsymbol{\omega} - \epsilon(2\lambda\boldsymbol{\omega} + \nabla_{\boldsymbol{\omega}} L(\mathbf{f}(\boldsymbol{x}^t;\boldsymbol{\theta}), \mathbf{y}^t))\\ \boldsymbol{\beta} &\leftarrow \boldsymbol{\beta} - \epsilon\nabla_{\boldsymbol{\beta}} L(\mathbf{f}(\boldsymbol{x}^t;\boldsymbol{\theta}), \mathbf{y}^t),\end{aligned}$$

where $\boldsymbol{\beta} = (\mathbf{b}, \mathbf{c})$ contains the offset parameters, ϵ is the learning rate and t is incremented after each training example, modulo n.

Non-linear Transformations Support Linear Classification

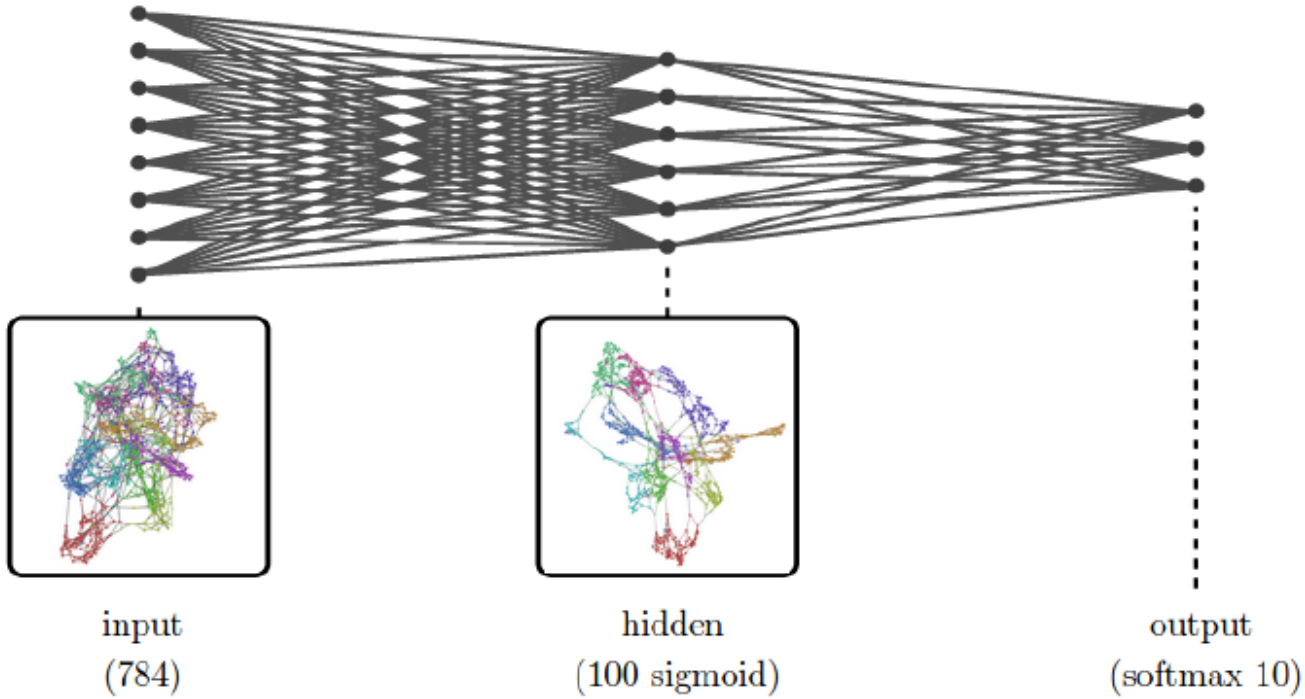
An Example



The red and blue solid curves are where the training examples come from. When the data is properly transformed non-linearly by the first hidden layer, the good decision surface becomes a linear one, which can be implemented by the output layer

Non-linear Transformations (cont'd)

Intuition via Dimensionality Reduction



784 inputs (pixels of a 28×28 image of a digit), 100 hidden units and 10 outputs corresponding to the 10 digit classes.

The squares below the input and hidden layer show where the training examples are in the reduced space, with one point per example, colored according to the digit class.

Estimating Conditional Statistics

We can generalize linear regression to regression via any function f by defining the mean squared error of f:

$$\mathbb{E}||\mathbf{y} - f(\boldsymbol{x})||^2$$

where the expectation is over the training set during training, and over the data generating distribution to obtain generalization error. We can generalize its interpretation beyond the case where f is linear or affine, uncovering an interesting property:

$$\arg\min_{f \in \mathbb{H}} \mathbb{E}||\mathbf{y} - f(\boldsymbol{x})||^2 = \mathbb{E}[\mathbf{y}|\boldsymbol{x}]$$

provided that our set of functions $\mathbb{H}$ contains $\mathbb{E}[\mathbf{y}|\boldsymbol{x}]$.

Family of Input-Output Functions

Multi-layer neural networks compose simple transformations in order to obtain highly non-linear ones. Here we show how to construct an MLP by choosing the hyperbolic tangent activation function instead of the sigmoid activation function:

$$\mathbf{h}^k = \tanh(\mathbf{b}^k + \mathbf{W}^k \mathbf{h}^{k-1}),$$

where $\mathbf{h}^0 = \boldsymbol{x}$ is the input of the neural net, $\mathbf{h}^k, k > 0$ is the output of the k-th hidden layer, which has weight matrix $\mathbf{W}^k$ and offset (or bias) vector $\mathbf{b}^k$.

If we want the output $\mathbf{f}(\boldsymbol{x}; \boldsymbol{\theta})$ to lie in some desired range, then we typically define an output layer non-linearity. The output layer is generally different from the tanh, depending on the type of output to be predicted and the associated loss function.

Non-Linear Neural Network Activation Functions

Beyond the Sigmoid and the Hyperbolic Tangent Activation Functions

Non-linear neural network activation functions are typically combined with an affine transformation $\mathbf{a} = \mathbf{b} + \mathbf{W}\boldsymbol{x}$ and applied element-wise:

$$\mathbf{h} = \phi(\mathbf{a}) \Leftrightarrow h_i = \phi(a_i) = \phi(b_i + \mathbf{W}_i.\boldsymbol{x}).$$

- **Rectifier** or **Rectified Linear Unit** (**ReLU**) or **Positive Part**: transformation of the output of the previous layer: $\phi(\mathbf{a}) = \max(0, \mathbf{a})$, also written $\phi(\mathbf{a}) = \mathbf{a}^+$.
- **Softmax**: vector-to-vector transformation

$$\phi(\mathbf{a}) = \text{softmax}(\mathbf{a}) \Leftrightarrow [\phi(\mathbf{a})]_i = e^{a_i} / \sum_j e^{a_j}$$

such that $\sum_i [\phi(\mathbf{a})]_i = 1$ and $[\phi(\mathbf{a})]_i > 0$, i.e., the softmax output can be considered as a probability distribution over a finite set of outcomes.

Non-Linear Neural Network Activation Functions (cont'd)

Beyond the Sigmoid and the Hyperbolic Tangent Activation Functions

- **Radial Basis Function** (**RBF**) unit: acts on $\boldsymbol{x}$ using $\mathbf{h}_i = \exp(-||\mathbf{w}_i - \boldsymbol{x}||^2/\sigma_i^2)$ that corresponds to template matching. Example: $\mathbf{w}_i = \boldsymbol{x}^t$ for some assignment of samples t to hidden unit templates i.
- **Softplus**: $\phi(a) = \zeta(a) = \log(1 + e^a)$.
- **Hard tanh**: this is shaped similarly to the tanh and the rectifier but unlike the latter, it is bounded, $\phi(a) = \max(-1, \min(1, a))$.
- **Absolute Value Rectification**: $\phi(a) = |a|$.
- **Maxout**: It generalizes the rectifier but introduces multiple weight vectors $\mathbf{w}_i$ (called filters) for each hidden unit: $h_i = \max_i(b_i + \mathbf{w}_i \boldsymbol{x})$.

What is Back-Propagation?

Back-propagation is a method for computing gradients for multi-layer neural networks. The method can easily be generalized to arbitrary families of parametrized functions and their corresponding computational graph.

In machine learning the output of the function to differentiate (e.g., the training criterion J) is a scalar. We are interested in its derivative with respect to a set of parameters (considered to be the elements of a vector $\boldsymbol{\theta}$), or equivalently, a set of inputs.

The partial derivative of J with respect to $\boldsymbol{\theta}$ (called the gradient) tells us whether $\boldsymbol{\theta}$ should be increased or decreased in order to decrease J, and is a crucial tool in optimizing the training objective.

What is Back-Propagation? (cont'd)

It can be readily proven that the back-propagation algorithm for computing gradients has optimal computational complexity in the sense that there is no algorithm that can compute the gradient faster (in the O sense, i.e., up to an additive and multiplicative constant).

The basic idea of the back-propagation algorithm is that the partial derivative of the cost J with respect to parameters $\boldsymbol{\theta}$ can be decomposed recursively by taking into consideration the composition of functions that relate $\boldsymbol{\theta}$ to J, via intermediate quantities that mediate that influence, e.g., the activations of hidden units in a deep neural network.

Chain Rule for Taking Derivatives

Brief Review

Let $z = z(y(x))$ be a composite function. To compute the derivative, we use the chain rule:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \quad \text{or} \quad dz = (\frac{dz}{dy})(\frac{dy}{dx})dx.$$

If $z = z(y_1(x), y_2(x), ..., y_n(x))$ then

$$\frac{dz}{dx} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i}\frac{dy_i}{dx}.$$

If $z = z(y(x_1, x_2, ..., x_n))$ then

$$\nabla z = \frac{dz}{dy}(\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, ..., \frac{\partial y}{\partial x_n})^T,$$

Chain Rule for Taking Derivatives

Cost Function in Machine Learning

The "output" cost or objective function is $J = J(g(\boldsymbol{\theta}))$ and we are interested in the gradient with respect to the parameters $\boldsymbol{\theta}$. The intermediate quantities $g(\boldsymbol{\theta})$ correspond to neural network activations. The gradient of the cost function amounts to

$$\nabla_{\boldsymbol{\theta}} J(g(\boldsymbol{\theta})) = \nabla_{g(\boldsymbol{\theta})} J(g(\boldsymbol{\theta})) \frac{\partial g(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$$

which works also when J, g or $\boldsymbol{\theta}$ are vectors rather than scalars (in which case the corresponding partial derivatives are understood as Jacobian matrices of the appropriate dimensions).
If g is a vector, we can rewrite the above as follows:

$$\nabla_{\boldsymbol{\theta}} J(g(\boldsymbol{\theta})) = \sum_i \frac{\partial J(g(\boldsymbol{\theta}))}{\partial g_i(\boldsymbol{\theta})} \frac{\partial g_i(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}.$$

Forward Propagation Computation

We consider an MLP with affine layers composed with an arbitrary elementwise differentiable (almost everywhere) non-linearity f.

There are l layers, each mapping their vector-valued input $\mathbf{h}^{k-1}$ to a pre-activation vector $\mathbf{a}^k$ via a weight matrix $\mathbf{W}^k$ which is then transformed via f into $\mathbf{h}^k$.

The input vector $\boldsymbol{x}$ corresponds to $\mathbf{h}^0$ and the predicted outputs $\hat{\mathbf{y}}$ corresponds to $\mathbf{h}^l$.

The loss $L(\hat{\mathbf{y}}, \mathbf{y})$ depends on the output $\hat{\mathbf{y}}$ and on a target $\mathbf{y}$. The loss may be added to a regularizer Ω to obtain the cost function J.

Algorithm 1: Forward Propagation Computation

Algorithm 1 Forward propagation computation in matrix form for a multi-layer neural network with l affine layers and a non-linearity.

$\mathbf{h}^0 = \boldsymbol{x}$
for $i = 1, ..., l$ **do**
  $\mathbf{a}^k = \mathbf{b}^k + \mathbf{W}^k \mathbf{h}^{k-1}$,
  $\mathbf{h}^k = f(\mathbf{a}^k)$.
end for
$\hat{\mathbf{y}} = \mathbf{h}^l$
$J = L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \Omega$

Each unit at layer k computes an output h_i^k as follows:

$$a_i^k = b_i^k + \sum_j W_{ij}^k h_j^{k-1} \quad \text{and} \quad h_i^k = f(a_i^k).$$

Note that the forward computation uses the input $\mathbf{x}$.

Back-Propagation in an MLP

We consider an MLP with multiple hidden layers and will recursively apply the chain rule for computing derivatives.

The algorithm proceeds by first computing the gradient of the cost J with respect to output units, and these are used to compute the gradient of J with respect to the top hidden layer activations, which directly influence the outputs. We can then continue computing the gradients of lower level hidden units one at a time in the same way.

The gradients on hidden and output units can be used to compute the gradient of J with respect to the parameters. In a typical network divided into many layers with each layer parameterized by a weight matrix and a vector of biases, the gradient on the weights and the biases is determined by the input to the layer and the gradient on the output of the layer.

The Algorithm Behind Back-Propagation

We do not know what the hidden units ought to do, but we can compute how fast the error changes as we change a hidden activity.

- Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities;
- Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined.

We can compute error derivatives for all the hidden units efficiently at the same time.

- Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit.

The Algorithm Behind Back-Propagation (cont'd)

First convert the discrepancy between each output and its target value into an error derivative:

$$\frac{\partial J}{\partial a_j^l} = \frac{\partial J}{\partial h_j^l} \cdot \frac{\partial h_j^l}{\partial a_j^l}.$$

Here $\frac{\partial J}{\partial h_j^l}$ is the derivative of the cost function with respect to the output. The second term on the RHS amounts to f'.

Then compute error derivatives in each hidden layer from error derivatives in the layer above, i.e., compute

$$\frac{\partial J}{\partial h_i^k} = \sum_j \frac{\partial J}{\partial a_j^l} \frac{\partial a_j^l}{\partial h_i^k} = \sum_j W_{ij}^l \frac{\partial J}{\partial a_j^l}.$$

The Algorithm Behind Back-Propagation (cont'd)

Then use error derivatives w.r.t. activities to get error derivatives w.r.t. the incoming weights:

$$\frac{\partial J}{\partial W_{ij}^k} = \frac{\partial J}{\partial a_j^l} \frac{\partial a_j^l}{\partial W_{ij}^k} = h_i^k \frac{\partial J}{\partial a_j^l}.$$

Note that since the l's is the output layer, we used the notation $k = l - 1$ for the hidden layer just before the output layer.

Algorithm 2: Back-Propagation

Algorithm 2 Backward computation of the gradients of the cost function with respect to the parameters for each layer k.

Compute the gradient on the output layer

$\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} J = \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \nabla_{\hat{\mathbf{y}}} \Omega$

for $k = l, l-1, ..., 1$ **do**

Convert the gradient on the layers output into a gradient into the pre-nonlinearity activation

$\mathbf{g} \leftarrow \nabla_{\mathbf{a}^k} J = \mathbf{g} \odot f'(\mathbf{a}^k)$

Compute gradients on weights and biases

$\nabla_{\mathbf{b}^k} J = \mathbf{g} + \lambda \nabla_{\mathbf{b}^k} \Omega$

$\nabla_{\mathbf{W}^k} J = \mathbf{g}\, \mathbf{h}^{(k-1)T} + \lambda \nabla_{\mathbf{W}^k} \Omega$

Propagate gradients w.r.t. the next lower-level hidden layers activations

$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{k-1}} J = \mathbf{W}^{(k)T} \mathbf{g}$

end for

A Computational Graph Model

Here we model the evaluation of the cost function J on a graph called the computational graph. Each node u_i of the graph denotes a numerical quantity that is obtained by performing a computation requiring the values u_j of other nodes, with $j < i$. The nodes satisfy a partial order which dictates in what order the computation can proceed.

We will use the generic notation $u_i = f_i(\mathbf{a}^i)$, where $\mathbf{a}^i$ is a list of arguments for the application of f_i to the values u_j of the parents of i in the graph: $\mathbf{a}^i = (u_j)_{j \in \mathsf{parents}(i)}$.

Algorithm 3: Forward Propagation Computation

Algorithm 3 A computational graph describing forward propagation. Each node computes numerical value u_i by applying a function f_i to its argument list $\mathbf{a}^i$ that comprises the values of previous nodes $u_j, j < i$, with $j \in \text{parents}(i)$. The input to the computational graph is the vector $\boldsymbol{x}$, and is set into the first m nodes u_1 to u_m. The output of the computational graph is read off the last node u_n.

for $i = 1, ..., m$ **do**
  $u_i \leftarrow x_i,$
end for
for $i = m + 1, ..., n$ **do**
  $\mathbf{a}^i \leftarrow (u_j)_{j \in \text{parents}(i)}$
  $u_i \leftarrow f_i(\mathbf{a}^i)$
end for
return u_n

Direct and Indirect Effects in a Computational Graph

An Example

We consider $f_3(a_{31}, a_{32}) = e^{a_{31}+a_{32}}$ and $f_2(a_{21}) = a_{21}^2$, while $u_3 = f_3(u_2, u_1)$ and $u_2 = f_2(u_1)$. The direct derivative of f_3 with respect to its argument a_{32} (keeping a_{31} fixed) is $\frac{\partial f_3}{\partial a_{32}} = e^{a_{31}+a_{32}}$.

On the other hand, if we consider the variables u_3 and u_1 to which these arguments correspond, there are two paths from u_1 to u_3, and we obtain as the total derivative the sum of partial derivatives over these two paths, $\frac{\partial u_3}{\partial u_1} = e^{u_1+u_2}(1 + 2u_1)$.

The results are different because $\frac{\partial u_3}{\partial u_1}$ involves not just the direct dependency of u_3 on u_1 but also the indirect dependency through u_2.

Algorithm 4: Back-Propagation on a Computational Graph

Algorithm 4 $\pi(i,j)$ is the index of u_j as an argument to f_i. The back-propagation algorithm efficiently computes $\frac{\partial u_n}{\partial u_i}$ for all is (traversing the graph backwards), and in particular we are interested in the derivatives of the output node u_n with respect to the inputs $u_1, ..., u_m$ (which could be the parameters, in a learning setup). The cost of the overall computation is proportional to the number of arcs in the graph, assuming that the partial derivative associated with each arc requires a constant time.

$\frac{\partial u_n}{\partial u_n} \leftarrow 1$
for $j = n-1$ down to 1 **do**
  $\frac{\partial u_n}{\partial u_j} \leftarrow \sum_{i:j \in \text{parents}(i)} \frac{\partial u_n}{\partial u_i} \frac{\partial f_i(a_i)}{\partial a_{i,\pi(i,j)}}$
end for
return $(\frac{\partial u_n}{\partial u_i})_{i=1}^m$