ECBM E6040 Neural Networks and Deep Learning

Lecture #7: Feedforward Deep Networks (cont'd) and, Regularization of Deep or Distributed Models

Aurel A. Lazar

Columbia University
Department of Electrical Engineering

March 1, 2016

Outline of Part I

Outline of Part II

Outline of Part III

Part I

Review of Previous Lecture

Topics Covered

- Multilayer Perceptrons from the 1980s
- Estimating Conditional Statistics
- Parametrizing a Learned Predictor
- Computational Graphs and Back-Propagation

Learning Objectives

- MLPs are realizations/implementations of compositions of multi-input and multi-output parametric functions.
- Examples of non-linear activation functions: sigmoid, tanh, softmax, radial basis functions, maxout, etc.
- Backpropagation: a method for computing gradients for multi-layer neural networks.
- 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.

Learning Objectives (cont'd)

The back-propagation 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. The BPA continues 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.

Part II

Feedforward Deep Networks (cont'd)

Implementing Stochastic Transformations

Traditional neural networks implement a deterministic transformation of some input variables $\boldsymbol{x}$. We can extend neural networks to implement stochastic transformations of $\boldsymbol{x}$ by simply augmenting the neural network with randomly sampled inputs $\boldsymbol{z}$.

The neural network can then continue to perform deterministic computation internally, but the function $f(\boldsymbol{x}, \boldsymbol{z})$ will appear stochastic to an observer who does not have access to $\boldsymbol{z}$. Provided that f is continuous and differentiable, we can then generally apply back-propagation as usual.

Let y be obtained by sampling the Gaussian distribution $\mathcal{N}(\mu, \sigma)$. We interpret the sampling process as the transformation

$$y = \mu + \sigma z,$$

where z is a Gaussian random variable $\mathcal{N}(0, 1)$.

Implementing Stochastic Transformations (cont'd)

We are now able to back-propagate through the sampling operation, by regarding it as a deterministic operation with an additional input z.

Back-propagation through the sampling operation enables us to incorporate the transformation into a larger graph and, thereby, compute the derivatives of the loss function of interest $J(y)$.

We can also introduce functions that shape the distribution, e.g., $\mu = f(\boldsymbol{x}; \boldsymbol{\theta})$ and $\sigma = g(\boldsymbol{x}; \boldsymbol{\theta})$ and use back-propagation through this functions to derive $\nabla_{\boldsymbol{\theta}} J(y)$.

Implementing Stochastic Transformations (cont'd)

More generally, using the representation

$$p(y|\boldsymbol{x};\boldsymbol{\theta}) = p(y|\boldsymbol{\omega}),$$

where $\boldsymbol{\omega}$ is a variable containing both parameters $\boldsymbol{\theta}$ and the inputs $\boldsymbol{x}$, we have

$$y = f(\boldsymbol{z}, \boldsymbol{\omega}),$$

where $\boldsymbol{z}$ is a source of randomness. Here, $\boldsymbol{\omega}$ must not be a function of $\boldsymbol{z}$, and $\boldsymbol{z}$ must not be a function of $\boldsymbol{\omega}$. This is often called the reparametrization trick.

In neural network applications, $\boldsymbol{z}$ is typically drawn from some simple distribution, such as a unit uniform or unit Gaussian distribution. Complex distributions are achieved by allowing the deterministic portion of the network to reshape its input.

The Universal Approximation Theorem

The Power and Limitations of the UAP

Theorem (Universal Approximation Theorem)

A feedforward network with a linear output layer and at least one hidden layer with any squashing activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units.

The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well.

The Universal Approximation Theorem (cont'd)
The Power and Limitations of the UAP

Any continuous function on a closed and bounded subset of $\mathbb{R}^n$ is Borel measurable and therefore may be approximated by a neural network.

A neural network may also approximate any function mapping from any finite dimensional discrete space to another. Interestingly, universal approximation theorems have also been proven for a wider class of non-linearities which includes the now commonly used rectified linear unit.

The Universal Approximation Theorem (cont'd)

The Power and Limitations of the UAP

Feedforward networks provide a universal system for representing functions: given a function, we can find a large MLP that approximates/represents the function.

Even if the MLP is able to represent a function, there are no guarantees that the training algorithm will be able to learn it. Learning can fail for two different reasons:

- the optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function;
- the training algorithm might choose the wrong function due to overfitting.

There is no universal procedure for examining a training set of specific examples and choosing a function that will generalize to points not in the training set.

The Universal Approximation Theorem (cont'd)

The Power and Limitations of the UAP

The UAT says that there exists a network large enough to achieve any degree of accuracy; it does not say, however, how large this network will be.

Unfortunately, in the worse case, an exponential number of hidden units may be required. This is easiest to see in the binary case: the number of possible binary functions on vectors $v \in \{0,1\}^n$ is 2^{2^n} and selecting one such function requires 2^n bits, which will in general require $\mathcal{O}(2^n)$ degrees of freedom.

Remark

A feedforward network with a single layer is sufficient to represent any function, but it may be infeasibly large and may fail to learn and generalize correctly. Both of these failure modes suggest the use of deeper models.

The Universal Approximation Theorem (cont'd)

The Power and Limitations of the UAP

From a representation learning point of view the learning with a deep architecture

- consists of discovering a set of underlying factors of variation that can in turn be described in terms of other, simpler underlying factors of variation;
- expresses the belief that the function we want to learn is a computer program consisting of multiple steps, where each step makes use of the previous steps output. These intermediate outputs are not necessarily factors of variation, but can instead be analogous to counters or pointers that the network uses to organize its internal processing.

Using deep architectures does indeed express a useful prior over the space of functions the model learns.

Improved Performance with Piecewise Linear Hidden Units

Recent performance improvements of deep neural networks can be attributed to

- increases in computational power;
- the size of datasets.

The main algorithmic improvements are due to

- the use of piecewise linear units, such as absolute value rectifiers and rectified linear units. Such units consist of two linear pieces and their behavior is driven by a single weight vector.
- using a rectifying non-linearity is the single most important factor in improving the performance of a recognition system among several different factors of neural network architecture design.

Improved Performance with Piecewise LHUs (cont'd)

For small datasets, using rectifying nonlinearities is even more important than learning the weights of the hidden layers. Random weights are sufficient to propagate useful information through a rectified linear network, allowing the classifier layer at the top to learn how to map different feature vectors to class identities.

Learning is far easier in deep rectified linear networks than in deep networks that have curvature or two-sided saturation in their activation functions.

Just as piecewise linear networks are good at propagating information forward, back-propagation in such a network is also piecewise linear and propagates information about the error derivatives to all of the gradients in the network.

Improved Performance with Piecewise LHUs (cont'd)

The use rectified linear units is strongly motivated by biological considerations. The half-rectifying non-linearity was intended to capture the following properties of biological neurons:

- for some inputs, biological neurons are completely inactive.
- for some inputs, a biological neurons output is proportional to its input.
- most of the time, biological neurons operate in the regime where they are inactive (e.g., they have sparse activations).

Remark

Piecewise linear units represent by far the most popular class of hidden units.

Part III

Regularization of Deep or Distributed Models

Regularization from a Bayesian Perspective

There is a deep connection between the Bayesian perspective on estimation and the process of regularization. Many forms of regularization can be given a Bayesian interpretation.

Given a dataset $(\boldsymbol{x}^1, ..., \boldsymbol{x}^m)$, the posterior $p(\boldsymbol{\theta}|\boldsymbol{x}^1, ..., \boldsymbol{x}^m)$ can be obtained by combining the data likelihood $p(\boldsymbol{x}^1, ..., \boldsymbol{x}^m|\boldsymbol{\theta})$ with the prior belief on the parameter encapsulated by $p(\boldsymbol{\theta})$:

$$\log p(\boldsymbol{\theta}|\boldsymbol{x}^1, ..., \boldsymbol{x}^m) = \log p(\boldsymbol{\theta}) + \sum_i \log p(\boldsymbol{x}^i|\boldsymbol{\theta}) + \text{constant}$$

where the constant is $-\log Z$, with the normalization constant Z that depends only on the data.

When maximizing over $\boldsymbol{\theta}$, this constant does not matter.

Regularization from a Bayesian Perspective (cont'd)

In the context of maximum likelihood learning, the introduction of the prior distribution plays the same role as a regularizer in that it can be seen as a term (the first one below)

$$\log p(\boldsymbol{\theta}|\boldsymbol{x}^1, ..., \boldsymbol{x}^m) = \log p(\boldsymbol{\theta}) + \sum_i \log p(\boldsymbol{x}^i|\boldsymbol{\theta}) + \text{constant}$$

added to the objective function that is added (to the second term, the log-likelihood) in hopes of achieving better generalization, despite of its detrimental effect on the likelihood of the training data.

Classical Regularization

We denote the regularized objective function by $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta}),$$

where α is a hyperparameter that weights the relative contribution of the norm penalty term, Ω, relative to the standard objective function $J(\boldsymbol{x}; \boldsymbol{\theta})$.

The hyperparameter α is a non-negative real number. Setting $\alpha = 0$ results in no regularization. Larger values of α correspond to more regularization.

Note that for neural networks, we typically use a parameter norm penalty Ω that only penalizes the interaction weights. The offsets are left unregularized. The offsets typically require less data to fit accurately than the weights.

L^2 Parameter Regularization

The L^2 parameter norm penalty commonly known as weight decay is given by $\Omega(\boldsymbol{\theta}) = \frac{1}{2}||\boldsymbol{w}||_2^2$.

As we will see, the L^2 regularization strategy drives the parameters closer to the origin. To simplify the presentation, we assume that $\boldsymbol{\theta} = \boldsymbol{w}$ (no offset term). The gradient of the total objective function amounts to

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \mathbf{X}, \boldsymbol{y}) = \alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \mathbf{X}, \boldsymbol{y}).$$

A single gradient step for updating the weights is

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon(\alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \mathbf{X}, \boldsymbol{y})),$$

or

$$\boldsymbol{w} \leftarrow (1 - \epsilon\alpha)\boldsymbol{w} - \epsilon \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \mathbf{X}, \boldsymbol{y})).$$

L^2 Parameter Regularization (cont'd)

To simplify the analysis, we consider a quadratic approximation to the objective function in the neighborhood of the empirically optimal value of the weights $\boldsymbol{w}^*$.

$$\hat{J}(\boldsymbol{w}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^T \mathbf{H}(\boldsymbol{w} - \boldsymbol{w}^*)$$

where $\mathbf{H}$ of J with respect to $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^*$.

There is no first order term in this quadratic approximation, because $\boldsymbol{w}^*$ is defined to be a minimum, where the gradient vanishes.

Likewise, because $\boldsymbol{w}^*$ is a minimum, we can conclude that $\mathbf{H}$ is positive semi-definite and

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \mathbf{H}(\boldsymbol{w} - \boldsymbol{w}^*).$$

L^2 Parameter Regularization (cont'd)

The location of the minimum of the regularized objective function is given by

$$\alpha \boldsymbol{w} + \mathbf{H}(\boldsymbol{w} - \boldsymbol{w}^*) = 0,$$

or

$$(\mathbf{H} + \alpha \mathbf{I})\boldsymbol{w} = \mathbf{H}\boldsymbol{w}^*,$$

or finally,

$$\tilde{\boldsymbol{w}} = (\mathbf{H} + \alpha \mathbf{I})^{-1}\mathbf{H}\boldsymbol{w}^*.$$

The regularization term moves the optimum from $\boldsymbol{w}^*$ to $\tilde{\boldsymbol{w}}$. As α approaches 0, $\tilde{\boldsymbol{w}}$ approaches $\boldsymbol{w}^*$.

L^2 Parameter Regularization (cont'd)

Here we investigate the case when α grows.
Because $\mathbf{H}$ is real and symmetric, we can decompose it into a diagonal matrix

$$\mathbf{H} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T$$

with $\mathbf{\Lambda}$ having real eigenvalues, $\mathbf{Q}$ orthonormal eigenvectors and $\mathbf{Q}^{-1} = \mathbf{Q}^T$.
We have

$$\begin{aligned}\tilde{\boldsymbol{w}} &= (\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T + \alpha\mathbf{I})^{-1}\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T\boldsymbol{w}^* \\ &= [\mathbf{Q}(\mathbf{\Lambda} + \alpha\mathbf{I})\mathbf{Q}^T]^{-1}\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T\boldsymbol{w}^* \\ &= \mathbf{Q}(\mathbf{\Lambda} + \alpha\mathbf{I})^{-1}\mathbf{\Lambda}\mathbf{Q}^T\boldsymbol{w}^*,\end{aligned}$$

and finally

$$\mathbf{Q}^T\tilde{\boldsymbol{w}} = (\mathbf{\Lambda} + \alpha\mathbf{I})^{-1}\mathbf{\Lambda}\mathbf{Q}^T\boldsymbol{w}^*$$

L^2 Parameter Regularization (cont'd)

$\mathbf{Q}^T \tilde{\boldsymbol{w}}$ is rotating our solution parameters $\tilde{\boldsymbol{w}}$ into the basis defined by the eigenvectors of $\mathbf{Q}$ of $\mathbf{H}$.

Consequently, the effect of weight decay is to rescale the coefficients of the eigenvectors. The i'th component is rescaled by a factor of $\frac{\lambda_i}{\lambda_i + \alpha}$.

Along the directions where the eigenvalues of $\mathbf{H}$ are relatively large, for example, where $\lambda_i >> \alpha$, the effect of regularization is relatively small. However, components with $\lambda_i << \alpha$ will be shrunk to have nearly zero magnitude.

L^1 Parameter Regularization

L^1 regularization on the model parameter $\boldsymbol{w}$ is defined by

$$\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w}||_1 = \sum_i |\boldsymbol{w}_i|,$$

that is, as the sum of absolute values of the individual parameters.

The gradient of the regularized objective function is

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \mathbf{X}, \boldsymbol{y}) = \beta \text{sign}(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \mathbf{X}, \boldsymbol{y})$$

where sign$(\boldsymbol{w})$ is the sign of $\boldsymbol{w}$ applied element-wise.

Not that the regularization contribution to the gradient no longer scales linearly with $\boldsymbol{w}$, but instead it is a constant factor with a sign equal to sign$(\boldsymbol{w})$.

L^1 Parameter Regularization (cont'd)

The gradient of the approximation is given by

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \mathbf{H}(\boldsymbol{w} - \boldsymbol{w}^*).$$

where $\mathbf{H}$ is the Hessian matrix of J with respect to $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^*$.

To gain insight, we assume that the Hessian is diagonal, that is, $\mathbf{H} = \text{diag}([\gamma_1, ..., \gamma_N])$, where each $\gamma_i > 0$.
With this assumption, the solution of the minimum of the L^1 regularized objective function decomposes into a system of equations of the form:

$$\tilde{J}(\boldsymbol{w}; \mathbf{X}, \boldsymbol{y}) = \frac{1}{2}\gamma_i(\boldsymbol{w}_i - \boldsymbol{w}_i^*)^2 + \beta|\boldsymbol{w}_i^*|.$$

L^1 Parameter Regularization (cont'd)

For each dimension i, the optimal solution is of the form

$$\boldsymbol{w}_i = \mathsf{sign}(\boldsymbol{w}_i^*) \max(|\boldsymbol{w}_i^*| - \frac{\beta}{\gamma_i}, 0).$$