GI01/4C55: Supervised Learning

9. Projection Pursuit and Neural Networks

December 5, 2005

Massimiliano Pontil

Today's plan

- Projection pursuit regression (PPR)
- Neural networks
- Relation to PPR
- Back-propagation algorithm
- Example: optical character recognition

Bibliography: These lecture notes are available at:
http://www.cs.ucl.ac.uk/staff/M.Pontil/courses/index-SL05.htm
Lecture notes are based on Hastie, Tibshirani & Friedman, Chapter 11.
See also Bishop, Chapter 4

PPR (I)

Projection pursuit regression (PPR) learns functions of the form

$$f(\mathbf{x}) = \sum_{n=1}^{N} g_n(\widehat{\mathbf{w}}_n \cdot \mathbf{x}) \qquad (1)$$

- $\widehat{\mathbf{w}}$ is a unit vector
- the function $g_n(\widehat{\mathbf{w}}_n \cdot \mathbf{x})$ from $\mathbb{R}^d$ to $\mathbb{R}$ is called **ridge function** (it varies only along the direction $\widehat{\mathbf{w}}_n$)
- the scalar variable $v_n := \widehat{\mathbf{w}}_n \cdot \mathbf{x}$ is the linear projection of $\mathbf{x}$ onto the unit vector $\widehat{\mathbf{w}}_n$

PPR builds a function as in (1) by iteratively learning the univariate functions g_n and the projection vectors $\widehat{\mathbf{w}}_n$

PPR (II)

- Can approximate continuous functions arbitrarily well (universal approximator) in a parsimonious way e.g.
$$x_1 x_2 = \frac{1}{4}\left\{(x_1 + x_2)^2 - (x_1 - x_2)^2\right\}$$
Higher order products can be represented similarly
- Interpretation of the fitted model is difficult (unlike CART which produces an easy to understand model of the data)

We will first discuss the PPR algorithm with $N = 1$ and then extend it to the general case

PPR (III)

If $N = 1$, we wish to minimize $\sum_{i=1}^m (y_i - g(\hat{\mathbf{w}} \cdot \mathbf{x}_i))^2$ over g and $\hat{\mathbf{w}}$. To this end, we choose an initial unit vector for $\hat{\mathbf{w}}$ and alternately compute $\hat{\mathbf{w}}$ and g until the error does not decrease more than a threshold

- Given the vector $\hat{\mathbf{w}}$ compute g by e.g. kernel ridge regression (say use a Gaussian kernel or polynomial kernel)

- Given g approximate $g(\hat{\mathbf{w}} \cdot \mathbf{x})$ by $g(\hat{\mathbf{w}}_{old} \cdot \mathbf{x}) + g'(\hat{\mathbf{w}}_{old} \cdot \mathbf{x})(\hat{\mathbf{w}} - \hat{\mathbf{w}}_{old}) \cdot \mathbf{x}_i$ and compute $\hat{\mathbf{w}}$ by weighted least squares regression:

$$\sum_{i=1}^m (y_i - g(\hat{\mathbf{w}} \cdot \mathbf{x}_i))^2 \approx \sum_{i=1}^m \left(g'(\hat{\mathbf{w}}_{old} \cdot \mathbf{x}_i)\right)^2 \left[\left(\hat{\mathbf{w}}_{old} \cdot \mathbf{x}_i + \frac{y_i - g(\hat{\mathbf{w}}_{old} \cdot \mathbf{x}_i)}{g'(\hat{\mathbf{w}}_{old} \cdot \mathbf{x}_i)}\right) - \hat{\mathbf{w}} \cdot \mathbf{x}_i\right]^2$$

PPR (IV)

$$\sum_{i=1}^m \left(y_i - \sum_{n=1}^N g_n(\hat{\mathbf{w}}_n \cdot \mathbf{x}_i)\right)^2$$

With more that one term in PPR model, the model is built by iteratively adding a pair $(g_n, \hat{\mathbf{w}}_n)$ each time:

$$\min_{\hat{w}_n, g_n} \left\{ \sum_{i=1}^m \left(y_i - \sum_{\ell=1}^{n-1} g_\ell(\hat{\mathbf{w}}_\ell \cdot \mathbf{x}_i) - g_n(\hat{\mathbf{w}}_n \cdot \mathbf{x}_i)\right)^2 \right\}$$

The algorithm stops when the next term does not appreciably improve the fit of the model. However, this may lead to overfitting and cross validation can be used to effectively choose N

Neural networks

Neural networks (NNets) (aka multi-layer perceptrons) implement a form of nonlinear function approximation schemes

A perceptron is a NNet with no hidden layers: $f(\mathbf{x}) = h(\mathbf{w} \cdot \mathbf{x})$

A 1-hidden layer NNet is a linear combination of perceptrons:

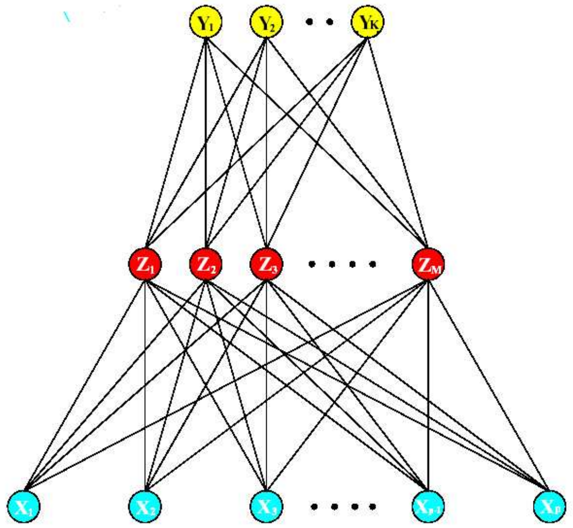
$$\sum_{n=1}^{N} u_n h(w_{0n} + \mathbf{w}_n \cdot \mathbf{x}) + u_0$$

This function depends nonlinearly on the parameters $u_0, u_n, w_{0n}, \mathbf{w}_n$.
We now describe a general one-hidden layer NNet with vectorial output ($\mathbf{y} \in \mathbb{R}^K$)

One-hidden layer NNet

NNets are typically represented by a network diagram

- Input features:
  $x_i,\ i = 1, \ldots, d$
- Inner layer features:
  $z_n = h(w_{0n} + \mathbf{w}_n \cdot \mathbf{x})$
  $n = 1, \ldots, N$
- Outputs layer:
  $f_k(\mathbf{x}) = g_k(\mathbf{t})$
  $t_k = u_{0k} + \mathbf{u}_k \cdot \mathbf{z},$
  $k = 1, \ldots, K$

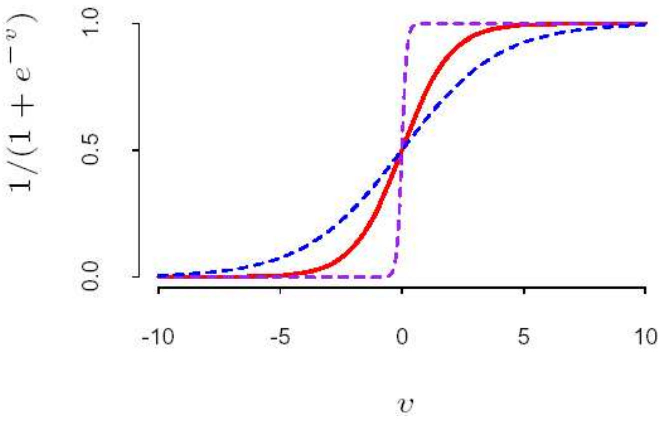


Parameters $\mathbf{w}_n, w_{0n}, \mathbf{u}_k, u_{0k}$ (often called **weights**) are learned
Functions h and g_k are prescribed (on page above $K = 1$ and $g_k(t) = t$)

Sigmoid function

The function h is called activation function and could be for example $h(v) = (1 + e^{-v})^{-1}$, used in logistic regression

Each perceptron (or unit) represents a neuron and the links (connections) the synapses in the brain. This is a mere speculation, however it has motivated early work on NNets



Note: Below we forget about the thresholds w_{0n}, u_{0k} (we can use the standard trick of adding one dimension to $\mathbf{x}$ and $\mathbf{z}$, e.g. $\mathbf{x} \to (1, \mathbf{x})$, and $\mathbf{w}_n \to (w_{0n}, \mathbf{w}_n)$)

Regression vs. classification

The output is computed as $f_k(\mathbf{x}) := g_k(\mathbf{t}(\mathbf{x}))$

- Classification: we choose g_k to be the **softmax** function

$$g_k(\mathbf{t}) = \frac{e^{t_k}}{\sum_{q=1}^{K} e^{t_q}}$$

  and use **Hamming coding**
  So if e.g. $K = 3$ we code the output as $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$
  For binary classification, $K = 2$, we can use one binary output only rather than $(0, 1)$ and $(1, 0)$

- Regression: we choose the identity function $g_k(\mathbf{t}) = t_k$ (when $K > 1$ this is called multiple output regression)

Relation to PPR

A 1-hidden layer NNet with g_k being the identity function is the same as PPR if we choose

$$g_n(\mathbf{w}_n \cdot \mathbf{x}) = u_n h(w_{0n} + \|\mathbf{w}_n\| \widehat{\mathbf{w}}_n \cdot \mathbf{x})$$

The ridge function g_n depends on three parameters u_n, w_{0n} and $\|\mathbf{w}_n\|$, so often 20/100 hidden units are used (as opposed to just 5 or so in PPR)

If the activation function h is replaced by a linear function the entire model collapses into a linear function of the input

Relation to kernel methods

- We can think of the hidden features z_n as a basis expansion of the inputs. Hence the NNets is essentially a linear function of these hidden features
- However, unline in kernel methods here the basis functions are learned from data (they are not prescribed in advance!)

We could also be tempted to think of $\sum_{n=1}^{N} u_n h(\mathbf{w}_n \cdot \mathbf{x})$ as a kernel expansion with $N = m$ and $\mathbf{w}_n = \mathbf{x}_i$, however the function $h(\mathbf{x}_i \cdot \mathbf{x})$ is not a valid kernel (it is not positive definite)!

Another popular choice for the activation function is a Gaussian, $\exp(\beta\|\mathbf{w} - \mathbf{x}\|^2)$, leading to **radial basis function networks**

Fitting NNets

The weight vectors $\mathbf{w}_n \in \mathbb{R}^{d+1}$, $n = 1, \ldots, N$ and $\mathbf{u}_k \in \mathbb{R}^{N+1}$, $k = 1, \ldots, K$ can be learned by minimizing the empirical error

$$E(\mathbf{f}) = \sum_{i=1}^{m} V(\mathbf{y}_i, \mathbf{f}(\mathbf{x}_i))$$

where $\mathbf{y}_i = (y_{i1}, \ldots, y_{iK})$, $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_K(\mathbf{x}))$
and $f_k(\mathbf{x}) = g_k\left(\sum_{n=1}^{N} u_n h(\mathbf{w}_n \cdot \mathbf{x})\right)$

We also use the notation $E_i = V(\mathbf{y}_i, \mathbf{f}(\mathbf{x}_i))$

Loss functions

$$E(\mathbf{f}) = \sum_{i=1}^{m} E_i = \sum_{i=1}^{m} V(\mathbf{y}_i, \mathbf{f}(\mathbf{x}_i))$$

- Classification:

$$V(\mathbf{y}_i, \mathbf{f}(\mathbf{x}_i)) = -\sum_{k=1}^{K} y_{ik} \log f_k(\mathbf{x}_i)$$

  With the softmax activation function this is similar to a logistic regression model

- Regression:

$$V(\mathbf{y}_i, \mathbf{f}(\mathbf{x}_i)) = \sum_{k=1}^{K} (y_{ik} - f_k(\mathbf{x}_i))^2$$

Learning algorithm

The parameters $\mathbf{w}_n$ and $\mathbf{u}_k$ are computed with a gradient method

$$\begin{aligned} \mathbf{w}_n^{(t+1)} &= \mathbf{w}_n^{(t)} - \eta^{(t)} \frac{\partial E}{\partial \mathbf{w}_n} \qquad (*) \\ \mathbf{u}_k^{(t+1)} &= \mathbf{u}_k^{(t)} - \eta^{(t)} \frac{\partial E}{\partial \mathbf{u}_k} \qquad (**) \end{aligned}$$

or the online version in which E is replaced by $E_{i(t)}$ where $i(t)$ is the example selected at time t (typically $i(t) = t \bmod m$). See also the discussion in Lecture Notes 3

Typically we do not want a global minimizer of E as it may overfit the data. We'll see later how to control overfitting

Computing the gradient

We analyze the updating rule for the square loss and $K = 1$
Recall that

$$\begin{aligned} f(\cdot) &= g(\mathbf{u} \cdot \mathbf{z}(\cdot)) \\ \mathbf{z}(\mathbf{x}) &= (h(\mathbf{w}_1 \cdot \mathbf{x}), \dots, h(\mathbf{w}_N \cdot \mathbf{x})) \end{aligned}$$

Using the chain rule for differentiation, we have that

$$\begin{aligned} \frac{\partial E_i}{\partial \mathbf{u}} &= -2(y_i - f(\mathbf{x}_i)) g'(\mathbf{u} \cdot \mathbf{z}_i) \mathbf{z}_i = \delta_i \mathbf{z}_i \\ \frac{\partial E_i}{\partial \mathbf{w}_n} &= -2(y_i - f(\mathbf{x}_i)) g'(\mathbf{u} \cdot \mathbf{z}_i) u_n h'(\mathbf{w}_n \cdot \mathbf{x}_i) \mathbf{x}_i = s_{ni} \mathbf{x}_i \end{aligned}$$

where we used the notation $\mathbf{z}_i = \mathbf{z}(\mathbf{x}_i)$. Note that

$$s_{ni} = h'(\mathbf{w}_n \cdot \mathbf{x}_i) u_n \delta_i \qquad \text{(back – propagation equation)}$$

Back-propagation algorithm

The update rules (*) and (**) can be implemented with a two-pass algorithm

- **forward pass:** compute the predicted values $f(\mathbf{x}_i)$ using the current value of the weights:

$$\mathbf{z}_i^{(t)} = \left(h(\mathbf{w}_n^{(t)} \cdot \mathbf{x}_i)\right)_{n=1}^{N}, \qquad f^{(t)}(\mathbf{x}_i) = g\left(\mathbf{u}^{(t)} \cdot \mathbf{z}_i^{(t)}\right)$$

- **backward pass:** the error δ_i is computed and then back-propagated to give the errors s_{ni}

Note the simplicity of the algorithm: each hidden unit passes and receives information only to and from units that share a connection with it (amenable to a parallel implementation)

Practical issues: initialization and preprocessing

- The error function E is not convex, hence the final solution depends on the starting value of the weights, $\mathbf{w}_n^{(0)}$ and $\mathbf{u}^{(0)}$
  - A starting weight near zero implies that initially the network behaves linearly (all sigmoid functions collapse into linear ones). The model then becomes nonlinear as weights increase (individual units introduce non-linearity when needed)
  - Starting with large weights leads to poor solutions
- It is good procedure to normalize (scale) the inputs to have zero mean and standard deviation one and choose the starting weights near zero (say in the range $[-0.5, 0.5]$)

Practical issues: overfitting

To avoid overfitting there are two standard procedures

- **Early stopping:** train the model for a while and stop before you reach a minimum (use a validation set to determine when to stop, e.g. compute validation error after each epoch of the network)

- **Weight decay:** add a penalty term to the error function which penalizes large values of the weights (similar to regularized least squares):

$$E(\mathbf{w},\mathbf{u}) + \lambda \left(\sum_{n=1}^{N} \|\mathbf{w}_n\|^2 + \sum_{k=1}^{K} \|\mathbf{u}_k\|^2 \right), \qquad \lambda > 0$$

More complex networks

More layers can be added recursively, e.g. we can go from a 1-hidden layer to a 2-hidden layer network by replacing each perceptron $\mathbf{w}_n \cdot \mathbf{x}$ by a 1-hidden layer network

$$\sum_{n=1}^{N} u_n h(\mathbf{w}_n \cdot \mathbf{x})$$

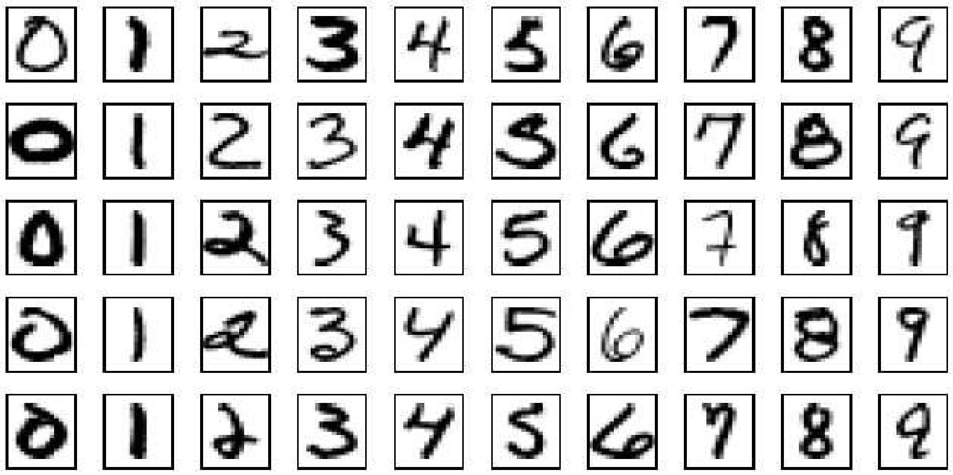
to

$$\sum_{n=1}^{N} u_n h \left(\sum_{q=1}^{Q} v_{nq} h(\mathbf{w}_{nq} \cdot \mathbf{x}) \right)$$

Practical issues: number of hidden unit/layers

- Better to have too many hidden units N than too few (appropriate regularization takes care of overfitting anyway)
- Typically N increases with the number of examples m and their dimension d
- Use of multiple hidden layers allows construction of hierarchical features at "different levels of resolution" (we illustrate this in the next example)
- for complex networks it may be advantageous to run the network multiple times and average the prediction over the multiple obtained networks (not average the weights!)

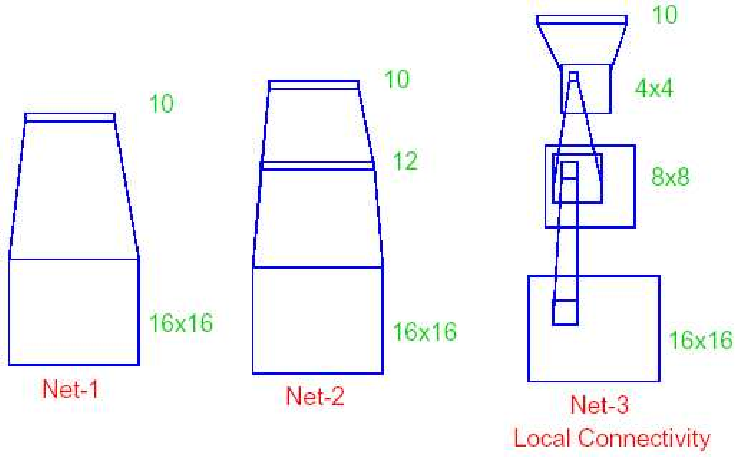
Example: handwritten digit classification



Examples of digits from U.S. Postal Service dataset (16 x 16 grayscale images)

Different NNets

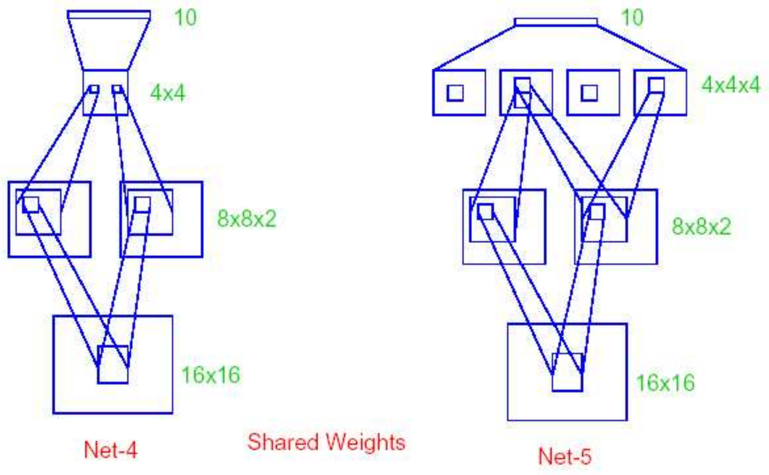
- Net-1: No hidden layer (equivalent to multinomial logistic regression)
- Net-2: 1 hidden layer, $N = 12$, fully connected
- Net-3: 2 hidden layers locally connected



Local connectivity: each hidden unit is connected to only a subset of the units below it (specifically, in Net-3 in the 1st hidden layer, each unit connects to a 3x3 patch of the input layer; in the second layer inputs are 5x5 patches

Different NNets (cont.)

- Net-4: 2 hidden layers locally connected with weight sharing
- Net-5: 2 hidden layers locally connected, two levels of weight sharing



Net-5 is motivated by the fact that features of handwriting style should appear in more than one part of a digit. The weight sharing implement a form hard constraint on the weights

Number of parameters

Network architecture	Links	Weights	Performance
Net-1: Single layer	2570	2570	80.0%
Net-2: 2-Layers	3214	3214	87.0%
Net-3: Locally connected	1226	1226	88.5%
Net-4: Constrained NNet 1	2266	1132	94.0%
Net-5: Constrained NNet 2	5194	1060	98.4%

Note that Net-4 and Net-5 have more links but fewer weights than Net-3 but better test performance