Neural Networks and Deep Learning

-Statistical Machine Learning-

Lecturer: Darren Homrighausen, PhD

High level overview

Neural networks are models for supervised learning

Linear combinations of features are passed through a non-linear transformation in successive layers

At the top layer, the resulting latent factors are fed into an algorithm for predictions

(Most commonly via least squares or logistic regression)

High level overview

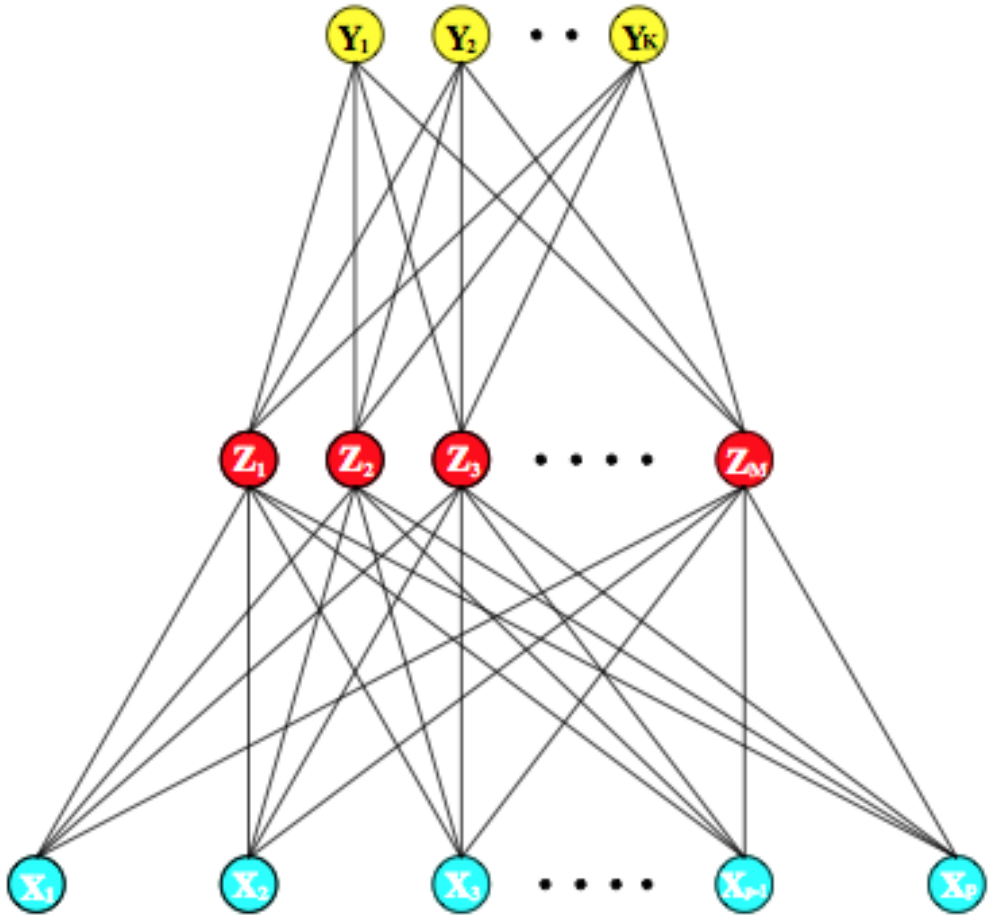


Figure: Single hidden layer neural network. Note the similarity to latent factor models

Nonparametric regression

Suppose $Y \in \mathbb{R}$ and we are trying to nonparametrically fit the regression function

$$\mathbb{E}Y|X = f_*(X)$$

A common approach (particularly when p is small) is to specify

- A fixed basis, $(\phi_j)_{j=1}^{\infty}$
- A tuning parameter J

Nonparametric regression

We follow this prescription:

1. Write[1]

$$f_*(X) = \sum_{j=1}^{\infty} \beta_j \phi_j(x)$$

   where $\beta_j = \langle f_*, \phi_j \rangle$

2. Truncate this expansion[2] at J

$$f_*^J(X) = \sum_{j=1}^{J} \beta_j \phi_j(x)$$

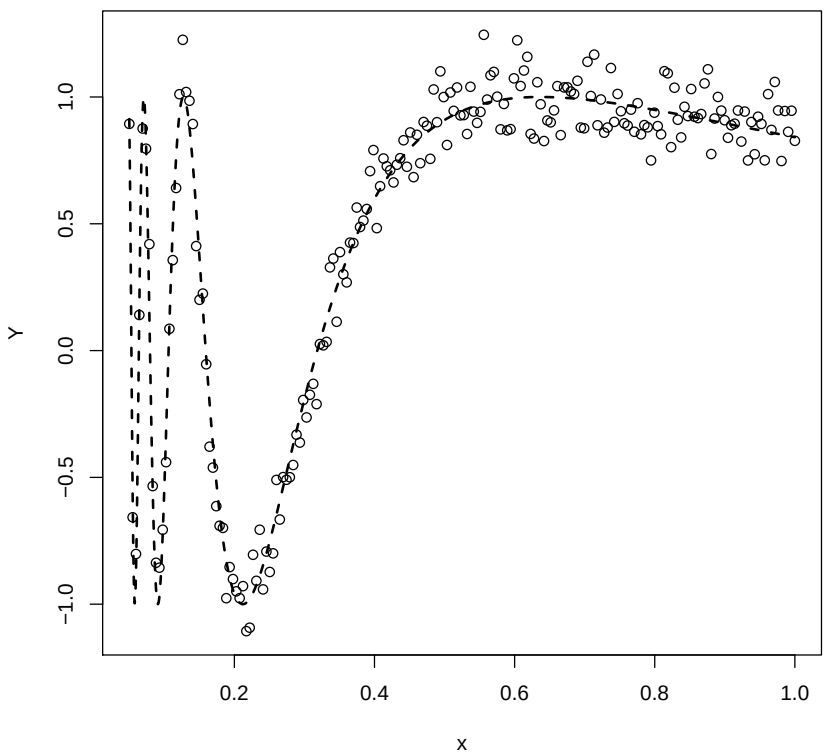
3. Estimate β_j with least squares

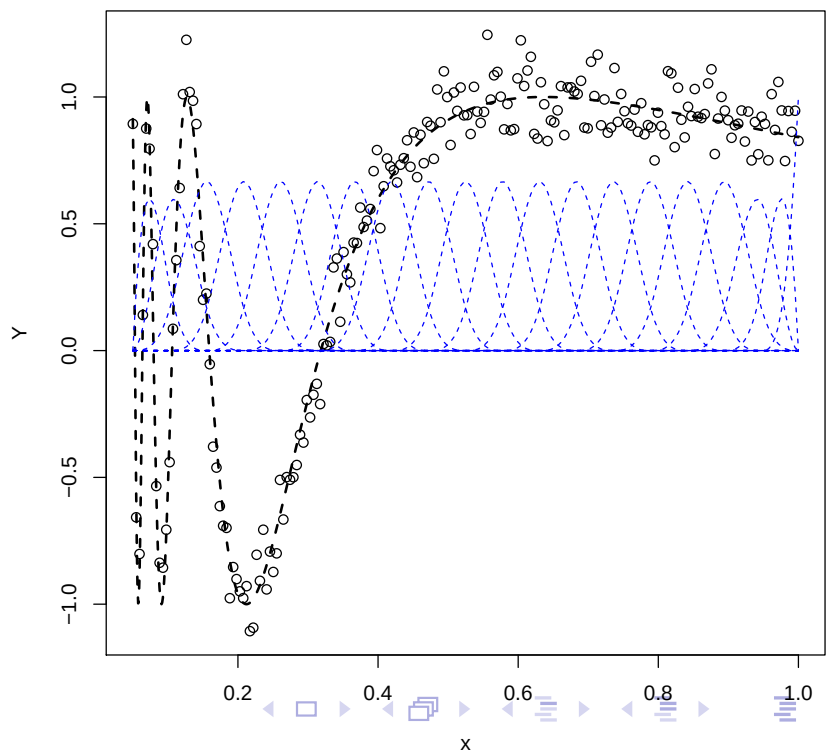
[1] Technically, f_* might not be in the span of the basis, in which case we have incurred an irreducible approximation error. Here, I'll just write f_* as the projection of f_* onto that span

[2] Often higher j are more rough $\Rightarrow$ this is a smoothness assumption

Nonparametric regression: Example

```
x = seq(.05,1,length=200)
Y = sin(1/x) + rnorm(100,0,.1)
plot(x,Y)
xTest = seq(.05,1,length=1000)
lines(xTest,sin(1/xTest),col='black',lwd=2,lty=2)
```



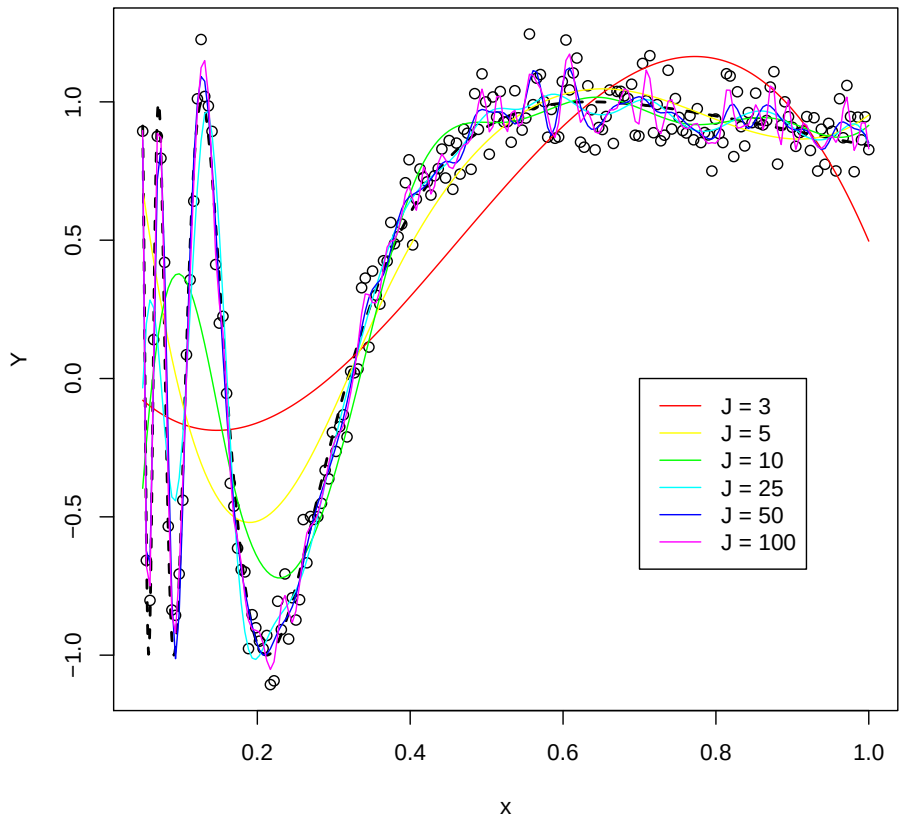


Nonparametric regression: Example

```
require(splines)
X = bs(x,df=20)
plot(x,Y)
lines(xTest,sin(1/xTest),col='black',lwd=2,lty=2)
matlines(x=x,X,lty=2,type='l',col='blue')
```

Nonparametric regression: Example

```
require(splines)
X    = bs(x,df=J)
Yhat = predict(lm(Y~.,data=X))
```



Nonparametric regression

The weaknesses of this approach are:

- The basis is fixed and independent of the data
- If p is large, then nonparametrics doesn't work well at all
  (See previous discussion on curse of dimensionality)
- If the basis doesn't 'agree' with f_*, then J will have to be large to capture the structure
- What if parts of f_* have substantially different structure?

An alternative would be to have the data tell us what kind of basis to use

High level overview

Letting $\mu(x) = \mathbb{E}Y|X = x$, and writing h as the link function, a simple[3] neural network can be phrased

$$h(\mu(x)) = \beta_0 + \sum_{j=1}^{J} \beta_j \sigma(\alpha_0 + \alpha^\top x)$$

(A particular nonlinear regression model, with basis functions σ)

Compare: Nonparametric regression would have the form

$$h(\mu(x)) = \beta_0 + \sum_{j=1}^{J} \beta_j \phi_j(x)$$

[3]Here simple indicates that there are much more complex versions, not that neural networks are basic in any way

High level overview

$$h(\mu(x)) = \beta_0 + \sum_{j=1}^{J} \beta_j \sigma(\alpha_{j0} + \alpha_j^\top x)$$

The main components are

- The derived features $Z_j = \sigma(\alpha_{j0} + \alpha_j^\top x)$ and are called the hidden units
  - The function σ is called the activation function and is very often $\sigma(u) = (1 + e^{-u})^{-1}$, known as the sigmoid function
  - The parameters α_j are estimated from the data.

    (This is the main difference from a basis expansion approach)
- The β are the coefficients of the regression
- The number of hidden units J is a tuning parameter

Activation function

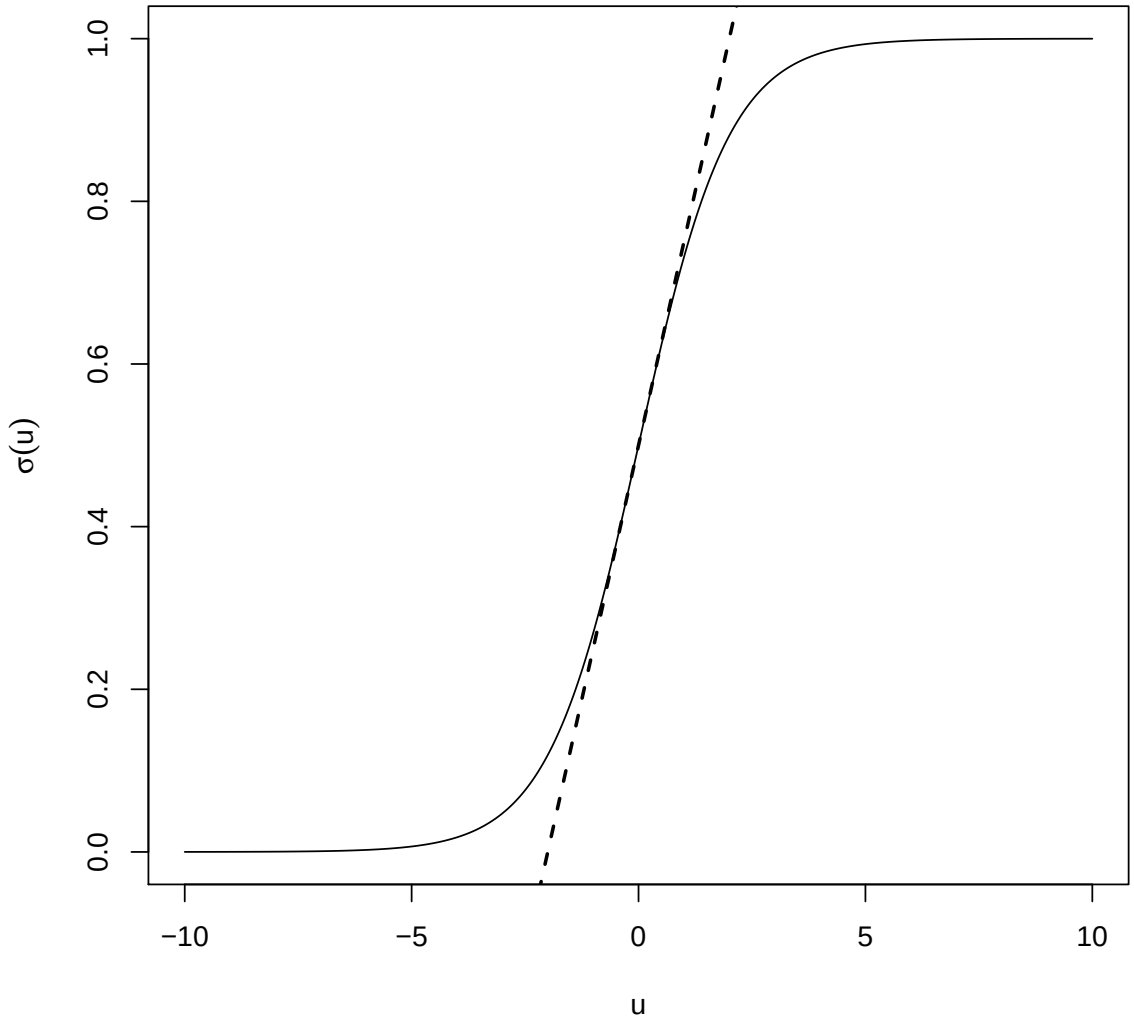
If $\sigma(u) = u$ is linear, then we recover classical methods

$$\begin{aligned} h(\mu(x)) &= \beta_0 + \sum_{j=1}^{J} \beta_j \sigma(\alpha_{j0} + \alpha_j^\top x) \\ &= \beta_0 + \sum_{j=1}^{J} \beta_j (\alpha_{j0} + \alpha_j^\top x) \\ &= \gamma_0 + \sum_{j=1}^{J} \gamma_j^\top x \end{aligned}$$

(The intercept terms are known as the bias terms)

If we look at a plot of the sigmoid function, it is quite linear near 0, but has nonlinear behavior further from the origin

Activation function



Hierarchical model

A neural network can be phrased as a hierarchical model

$$Z_j = \sigma(\alpha_{j0} + \alpha_j^\top X) \quad (j=1,\ldots J)$$
$$W_g = \beta_{g0} + \beta_g^\top Z \quad (g=1,\ldots G)$$
$$\mu_g(X) = h^{-1}(W_g)$$

The output depends on the application, where we map W_g to the appropriate space:

- Regression: The link function is $h(x) = x$ and we directly produce predictions of Y (here, $G = 1$)
- Classification: If there are G classes, we are modeling the probability of $Y = g$ and h is such that

$$\hat{\mu}_g(X) = \frac{e^{W_g}}{\sum_{g'=1}^{G} e^{W_{g'}}} \quad \text{and} \quad \hat{Y}(X) = \arg\max_g \hat{\mu}_g(X)$$

(This is called the softmax function for historical reasons)

Training neural networks

The neural networks have many unknown parameters

(They are usually called weights in this context)

These are

- α_{j0}, α_j for $j = 1, \ldots, J$ (total of $J(p+1)$ parameters)
- β_{g0}, β_g for $g = 1, \ldots, G$ (total of $G(J+1)$ parameters)

Total parameters: $\asymp Jp + GJ$

Training neural networks

The most common loss functions are

- Regression:

$$\hat{R} = \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

- Classification: Cross-entropy

$$\hat{R} = -\sum_{i=1}^{n} \sum_{g=1}^{G} Y_{ig} \log(f_g(X_i))$$

  - Here, Y_{ig} is an indicator variable for the g^{th} class. In other words $Y_i \in \mathbb{R}^G$

    (In fact, this means that Neural networks very seamlessly incorporate the idea of having multivariate response variables, even in regression)
  - With the softmax + cross-entropy, neural networks is a linear multinomial logistic regression model in the hidden units

Training neural networks

The usual approach to minimizing $\hat{R}$ is via gradient descent

This is known as back propagation

Due to the hierarchical form, derivatives can be formed using the chain rule and then computed via a forward and backward sweep

Neural networks: Back-propagation

For squared error, let $\hat{R}_i = (Y_i - \hat{Y}_i)^2$

Then

$$\frac{\partial \hat{R}_i}{\partial \beta_j} = -2(Y_i - \hat{Y}_i)Z_{ij}$$

$$\frac{\partial \hat{R}_i}{\partial \alpha_{jk}} = -2(Y_i - \hat{Y}_i)\beta_j \sigma'(\alpha_0 + \alpha_j^\top X_i)X_{ik}$$

Given these derivatives, a gradient descent update can be found

$$\hat{\beta}_j^{t+1} = \hat{\beta}_j^t - \gamma_t \sum_{i=1}^n \frac{\partial R_i}{\partial \beta_j}\bigg|_{\hat{\beta}_j^t}$$

(This later evaluation only matters if $h(x) \neq x$)

$$\hat{\alpha}_{jk}^{t+1} = \hat{\alpha}_{jk}^t - \gamma_t \sum_{i=1}^n \frac{\partial R_i}{\partial \alpha_{jk}}\bigg|_{\hat{\alpha}_{jk}^t}$$

(γ_t is called the learning rate, this needs to be set)

Neural networks: Back-propagation

Returning to

$$\frac{\partial \hat{R}_i}{\partial \beta_j} = -2(Y_i - \hat{Y}_i)Z_{ij} \qquad = a_i Z_{ij}$$

$$\frac{\partial \hat{R}_i}{\partial \alpha_{jk}} = -2(Y_i - \hat{Y}_i)\beta_j \sigma'(\alpha_0 + \alpha_j^\top X_i)X_{ik} \qquad = b_{ji} X_{ik}$$

Direct substitution of a_i into b_{ji} gives

$$b_{ji} = a_i \beta_j \sigma'(\alpha_0 + \alpha_j^\top X_i)$$

These are the back-propagation equations

Neural networks: Back-propagation

Back-propagation equations:

$$b_{ji} = a_i \beta_j \sigma'(\alpha_0 + \alpha_j^\top X_i)$$

The updates given by the gradient decent can be operationalized via a two-pass algorithm:

1. Forward pass: Current weights are fixed and predictions $\hat{Y}_i$ are formed
2. Backward pass: The a_i are computed, and then converted (aka back-propagated) to get b_{ji}
3. These updated quantities are used to take a gradient descent step

Neural networks: Back-propagation

Advantages:

- It's updates only depend on local information in the sense that if objects in the hierarchical model are unrelated to each other, the updates aren't affected

  (This helps in many ways, most notably in parallel architectures)
- It doesn't require second-derivative information
- As the updates are only in terms of R_i, the algorithm can be run in either batch or online mode

Down sides:

- It can be very slow
- Need to choose the learning rate γ_t

  (I don't know how to choose this well, do you?)

Neural networks: Other algorithms

There are a few alternative variations on the fitting algorithm

Many are using more general versions of non-Hessian dependent optimization algorithms

(For example: conjugate gradient)

The most popular are

- Resilient back-propagation (with or without weight backtracking)

  (Reidmiller (1994) and Riedmiller, Braun (1993))

- Modified globally convergent version

  (Anastasiadis et al. (2005))

Regularizing neural networks

As usual, we don't actually want the global minimizer of the training error (particularly since there are so many parameters)

Instead, some regularization is included

This is generated by a combination of

- a complexity penalization term
- early stopping on the back propagation algorithm used for fitting

  (This is related to the choice of starting values, so I'll defer this discussion for a few slides)

Regularizing neural networks

Explicit regularization comes in a couple of flavors

- Weight decay: This is like ridge regression in that we penalize the squared Euclidian norm of the weights

$$\sum \beta^2 + \sum \alpha^2$$

- Weight elimination: This encourages more shrinking of small weights

$$\sum \frac{\beta^2}{1+\beta^2} + \sum \frac{\alpha^2}{1+\alpha^2}$$

Common pitfalls

There are two areas to watch out for

- Nonconvexity: The neural network optimization problem is non convex. This makes any numerical solution highly dependant on the initial values. These must be
  - chosen carefully
  - regenerated several times to check sensitivity
- Scaling: Be sure to standardize the covariates before training
- Number of hidden units (J): It is generally better to have too many hidden units than too few (regularization can eliminate some). This includes adding multiple hidden layers

Starting values

The quality of the neural network predictions is very dependent on the starting values

As noted, the sigmoid function is nearly linearly near the origin.

Hence, starting values for the weights are generally randomly chosen near 0. Care must be chosen as:

- Weights equal to 0 will encode a symmetry that keeps the back propogation algorithm from changing solutions
- Weights that are large tend to produce bad solutions (overfitting)

This is like putting a prior on linearity and demanding the data add any nonlinearity

Starting values

Once several starting values + back-propogation pairs are run, we must sift through the output

Some common choices are:

- Choose the solution that minimizes training error
- Choose the solution that minimizes the penalized training error
- Average the solutions across runs

  (This is the recommended approach as it brings a model averaging/Bayesian flair)

Neural networks: General form

Generalizing to multi-layer neural networks, we can specify any number of hidden units:

(This is a heuristic representation and hence the indexing/notation is a bit vague. I'm eliminating the bias term for simplicity)

$$
\begin{aligned}
0 \text{ Layer} &:= \sigma(\alpha_0^\top X) \\
1 \text{ Layer:} &:= \sigma(\alpha_1^\top (0 \text{ Layer})) \\
&\vdots \\
\text{Top Layer} &:= \sigma(\alpha_{\text{Top}}^\top (\text{Top - 1 Layer})) \\
h(\mu_g(X)) &= \beta_{g0} + \beta_g^\top (\text{Top Layer}) \quad (g{=}1,\ldots G)
\end{aligned}
$$

Neural networks: General form

Some comments on adding layers:

- It has been shown (Hornik et al. (1989)) that one hidden layer is sufficient to approximate any piecewise continuous function
- However, this may take a huge number of hidden units (i.e. $J >> 1$)
- By including multiple layers, we can have fewer hidden units per layer. Also, we can encode (in)dependencies that can speed computations and localize the feature map

Neural networks: Example

Let's return to the doppler function example

We can try to fit it with a single layer NN with different levels of hidden units J

A notable difference with B-splines is that 'wiggliness' doesn't necessarily scale with J due to regularization

Some specifics:

- I used the R package neuralnet
- I regularized via a stopping criterion ($||\partial \ell||_\infty < 0.01$)
- I did 3 replications

Neural networks: Example

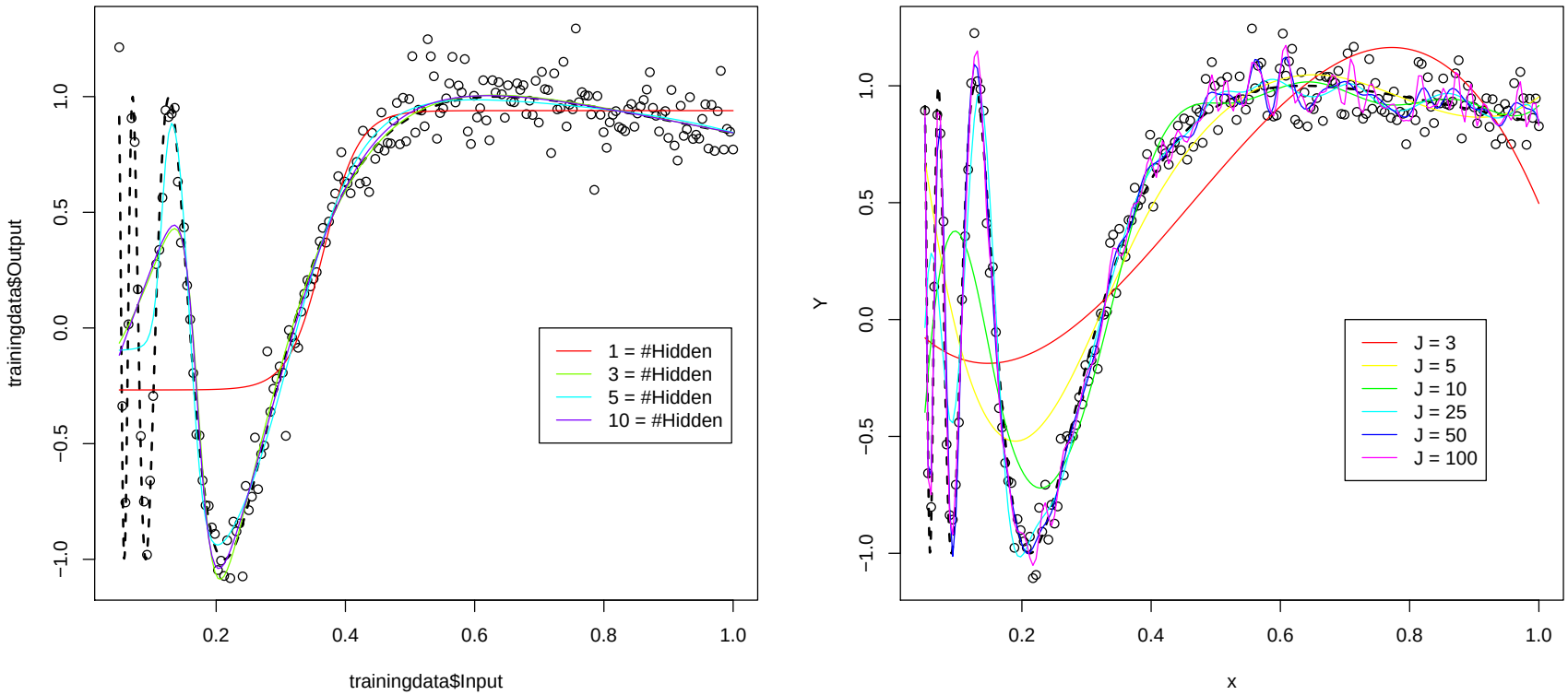


Figure: Single layer NN vs. B-splines

Neural networks: IMSE

$$\text{IMSE} = \int (\hat{f}(x) - f_*(x))^2 dx$$

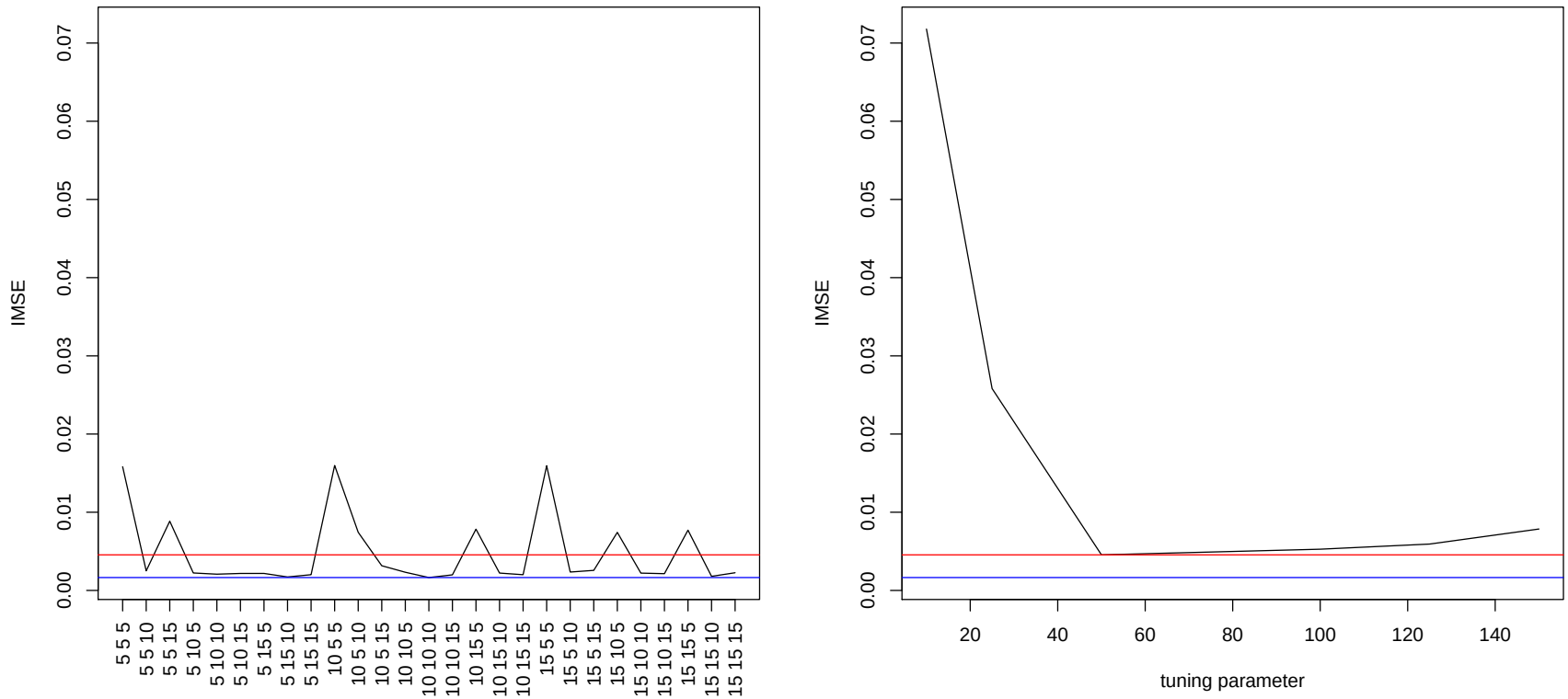


Figure: 3 layer NN[4] vs. B-splines

[4]The numbers mean (#(layer 1) #(layer 2) #(layer 3))

Neural networks: Example

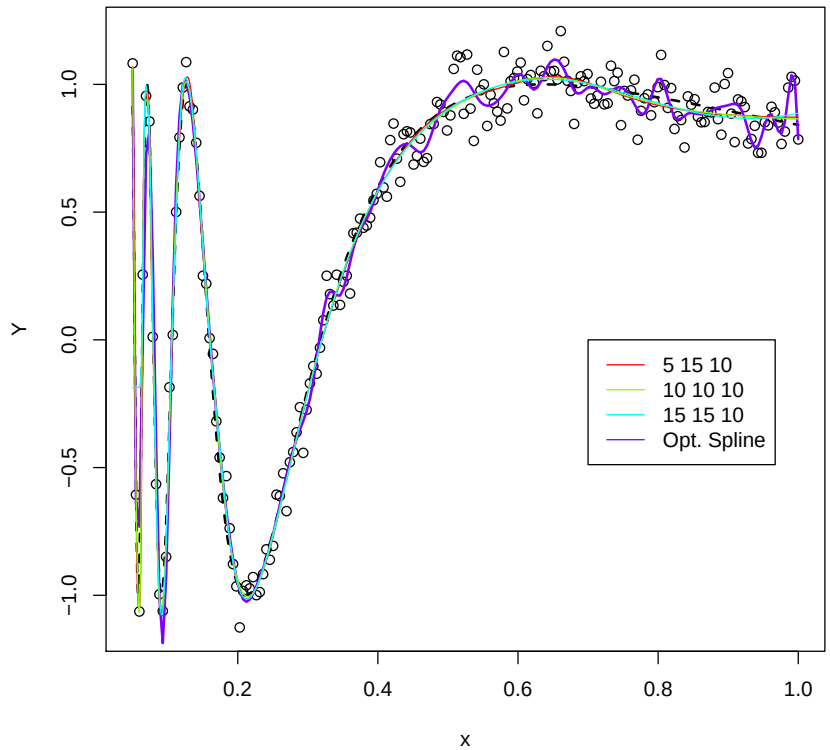


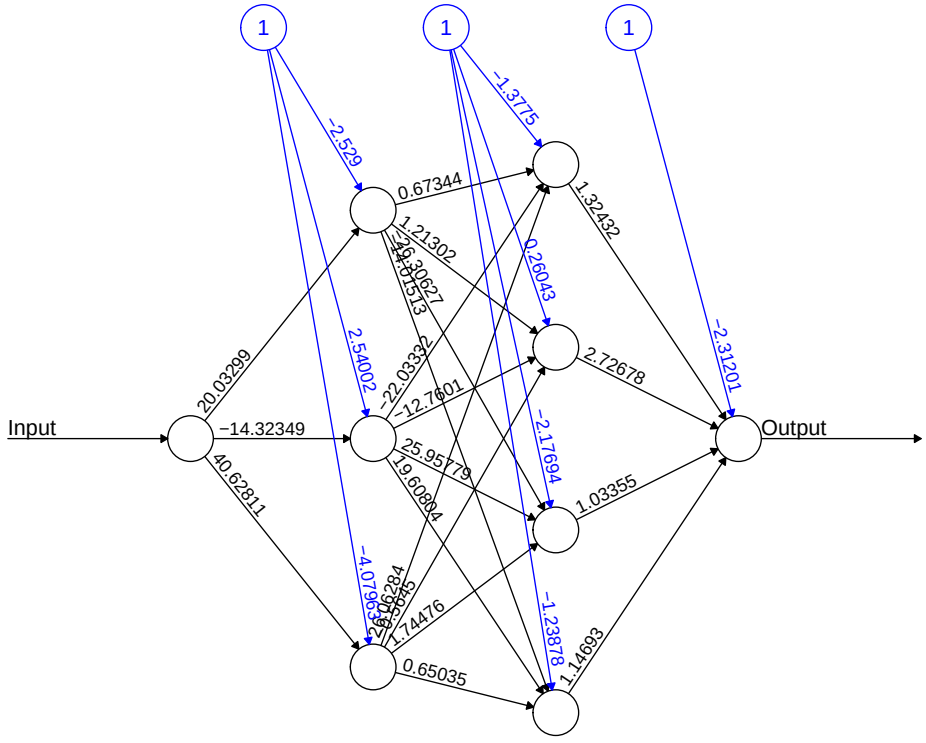
Figure: Optimal NNs vs. Optimal B-spline fit

Neural networks: Example

```
trainingdata  = cbind(x,Y)
colnames(trainingdata) = c("Input","Output")
testdata      = xTest

require("neuralnet")
J             = c(10,5,15)
nRep          = 3
nn.out        = neuralnet(Output~Input,trainingdata,
                          hidden=J, threshold=0.01,
                          rep=nRep)
nn.results = matrix(0,nrow=length(testdata),ncol=nRep)
for(reps in 1:nRep){
  pred.obj = compute(nn.out, testdata,rep=reps)
  nn.results[,reps] = pred.obj$net.result
}
Yhat = apply(nn.results,1,mean)
```

Neural networks: Example DAG[5]




```
nn.out = neuralnet(Output~Input,trainingdata,
                                hidden=c(3,4))
plot(nn.out)
```

[5]Directed acyclic graph

Neural networks: Localization

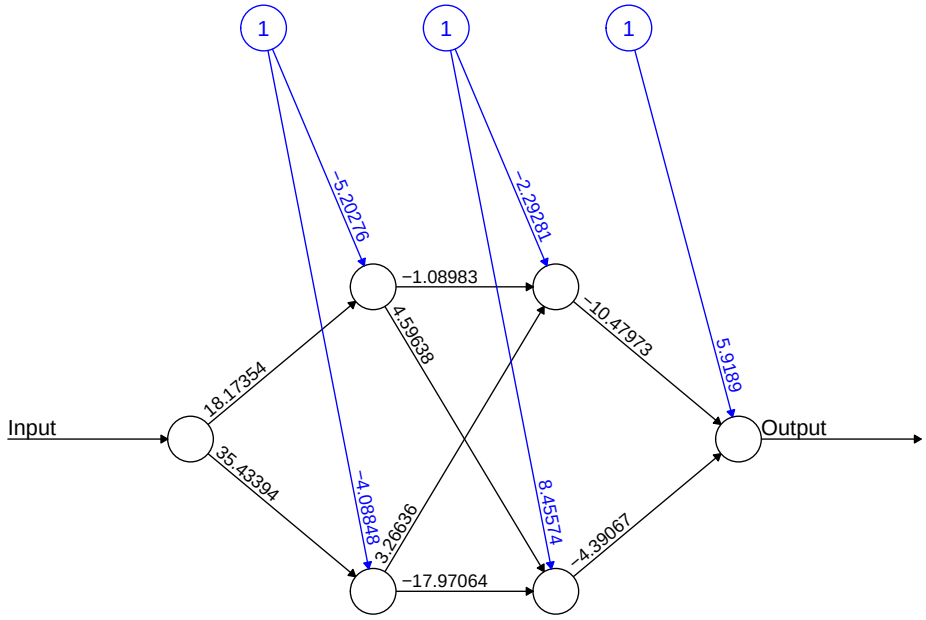
One of the main curses/benefits of neural networks is the ability to localize

This makes neural networks very customizable, but commits the data analyst to intensively examining the data

Suppose we are using 1 input and we want to restrict the implicit DAG

Neural networks: Localization

That is, we want to constrain some of the weights to 0



Error: 3.137653 Steps: 49829

Neural networks: Localization

We can do this in neuralnet via the exclude parameter

To use it, do the following:

```
exclude = matrix(1,nrow=2,ncol=3)
exclude[1,] = c(2,2,2)
exclude[2,] = c(2,3,1)
nn.out = neuralnet(Output~Input,trainingdata,
                   hidden=c(2,2), threshold=0.01,
                   exclude=exclude)
```

Neural networks: Localization

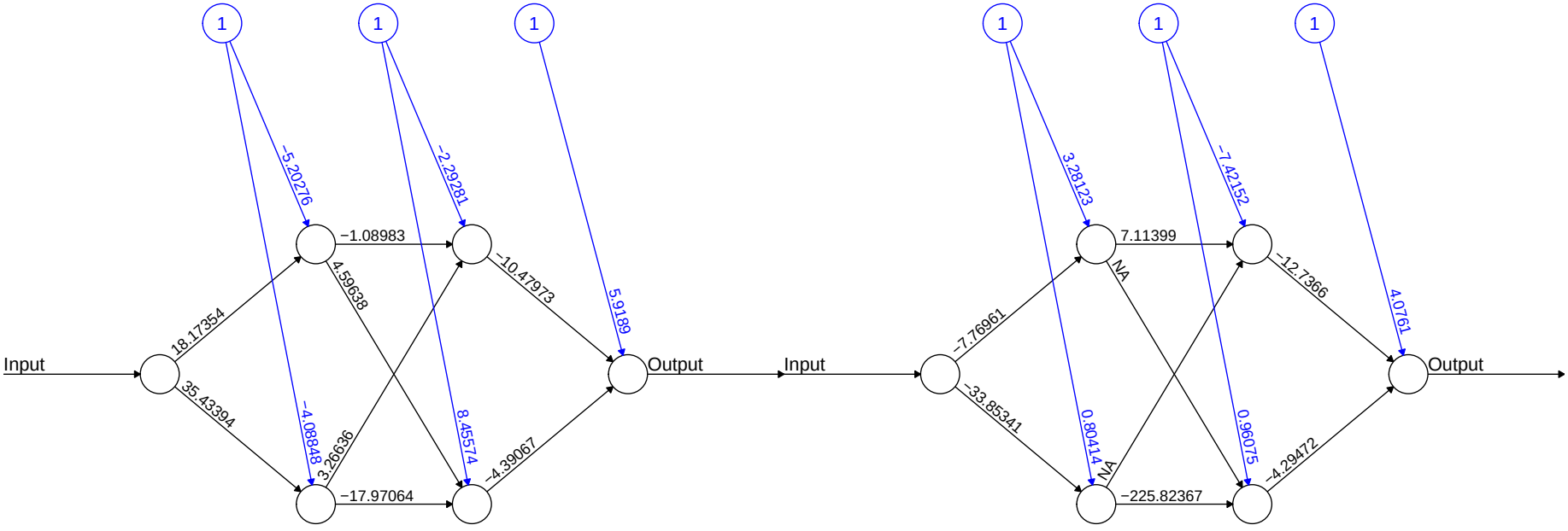
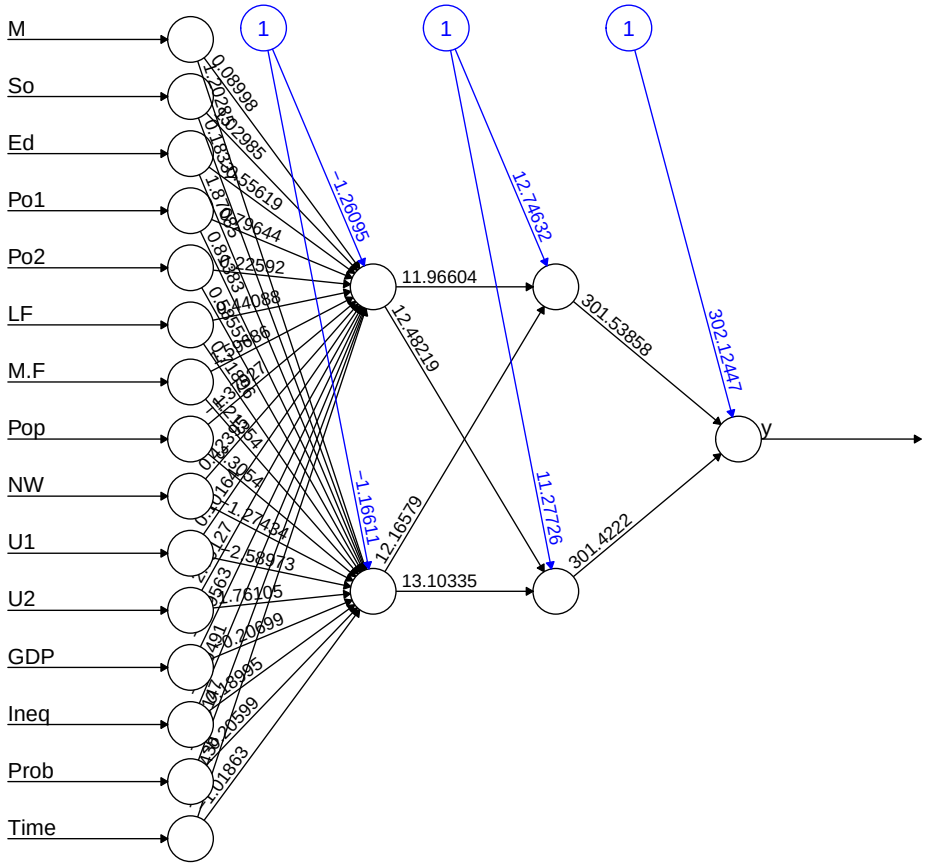


Figure: Not-constrained vs. constrained

Neural networks: Partial example

Suppose we are looking at crime data



Neural networks: Partial example

We may want to constrain the neural network to have neurons specifically about

- Demographic variables
- Police expenditure
- Economics

This type of prior information can be encoded via exclude

(This is, in my opinion, why neural networks do so well with functional-type data)

Projection pursuit

Projection pursuit

The projection pursuit idea came out of wanting to do nonparametrics in higher dimensions

For a covariate X, we form an additive model, but using univariate nonparametric smoothers of linear combinations of the covariates:

$$F(X) = \sum_{j=1}^{J} f_j(\alpha_j^\top X)$$

Projection pursuit

The scalar variables $Z_j = \alpha_j^\top X$ are the projections of the covariates onto the direction α_j

(Hence the name)

Any nonparametric smoother can be used for estimating f_j, however ones that gives easy derivative estimates (e.g. splines) are generally used

The f_j's can be fit via back fitting, but the weights α_j are generally only fit once

(See Friedman, Tukey (1974) for an early implementation and Friedman (1987) for an interesting application)

Projection pursuit

Neural networks are an extended/restricted version of projection pursuit

- Extended: Projection pursuit would correspond to a neural network with 1 layer
- Restricted: Projection pursuit allows for an arbitrary smoother of Z_i, whereas neural networks have a particular parametric form

Tuning parameters

Neural networks: Tuning parameters

The NN's from this example are quite similar, even with substantially different structure

Hence the degrees of freedom (df) of a NN is probably substantially less than the number of parameters

In fact, observe the PAC bound shown in Bartlett (1998):

Let

- $Y \in \{-1, 1\}$
- ℓ is squared error loss
- $\mathcal{F}$ be the set of single Layer NN with $1 \leq J \leq n$
- f_* be such that $\min_{f \in \mathcal{F}} \mathbb{P}\ell_f$
- $\hat{R}_\tau = \frac{1}{n}|\{i : Y_i f(X_i) < \tau\}|$
  (This is the training error with margin τ)
- $B \geq 1$ is a constant such that $||\beta||_1 \leq B$

Neural networks: Tuning parameters

Barlett (1998): With probability at least $1 - \eta$, for each $f \in \mathcal{F}$

$$\mathbb{P}\ell_f \leq \hat{R}_\tau + C\sqrt{\frac{1}{n}\left(\frac{B^2 p}{\tau^2}\log(B/\tau)\log^2(n) - \log(\eta)\right)}$$

Important:

- This PAC bound does not depend on J
- It does depend on the size of the coefficients (B)

Neural networks: Tuning parameters

The most common recommendation I've seen is to take the 3 tuning parameters: The number of hidden units, the number of layers, and the regularization parameter λ.

Either choose $\lambda = 0$ and use cross-validation to choose the number of hidden units

(This could be quite computationally intensive as we would need a reasonable 2-d grid over units $\times$ layers)

Or, fix a large number of layers and hidden units and cross-validate the tuning parameter λ

(This is the preferred method)

Neural networks: Tuning parameters

Due to Bartlett's result and related observations, the degrees of freedom are an appealing method for penalizing the training error

If we have an estimate of df, say $\hat{df}$, then we can report

$$\text{AIC} = \hat{\mathbb{P}} + 2\hat{df}\,\hat{\sigma}^2$$

Neural networks: Tuning parameters

Unfortunately, neuralnet provides a somewhat bogus measure of AIC/BIC

Here is the relevant part of the code

```
if (likelihood) {
  synapse.count = length(weights) - length(exclude)
  aic = 2 * error + (2 * synapse.count)
  bic = 2 * error + log(nrow(response))*synapse.count
}
```

They use the number of parameters for the degrees of freedom!

Neural networks: Tuning parameters

After doing a bit of a literature search, it appears the literature has been dominated by a series of papers by Ingrassia, Morlini

In Ingrassia, Morlini (2005), they propose to use the effective degrees of freedom

Their's is a perhaps overly simplistic take on the usual linear smoother intuition[6]

[6] I appear to have lost my slides outlining their approach. I don't have the heart to retype them now, so I'll just refer you to the very readable document "Neural Network Modeling for Small Datasets"

Neural networks: Tuning parameters

While writing up these notes, it strikes me that a reasonable research direction is to follow the approach we covered in the first homework: use Stein's method

We know that for any well behaved prediction algorithm $\hat{f}(X_i) = \hat{Y}_i$, that the degrees of freedom are

$$\frac{1}{n\sigma^2} \sum_{i=1}^{n} Cov(Y_i, \hat{Y}_i)$$

This equals

$$\mathbb{E} \nabla \cdot g(y),$$

where y is the response vector. This is known as the expected divergence

Challenge: Can you calculate this for neural networks?

Representation learning

Overview

Representation learning is the idea that performance of ML methods is highly dependent on the choice of data representation

For this reason, much of ML is geared towards transforming the data into the relevant features and then using these as inputs

This idea is as old as statistics itself, really,

(E.g. Pearson (1901), where PCA was first introduced)

However, the idea is constantly revisited in a variety of fields and contexts

Overview

Representation learning can be broken up into two philosophies

- Unsupervised: Here, we use the covariates only[7] to estimate what are hopefully relevant features of $p(X)$, the joint distribution of X

  (E.g. PCA, Laplacian eigenmaps, clustering, sparse coding, ...)

- Supervised: We form feature maps that take into account the nature of the joint distribution $p(X, Y)$

  (E.g. partial least squares, LDA, neural networks, linear regression)

[7] These methods are called semisupervised when these inputs are used to train a prediction algorithm

Overview

Commonly, these learned representations capture 'low level' information like overall shape types

Other sharp features, such as images, aren't captured

It is possible to quantify this intuition for PCA at least

Overview

Suppose the signals are the result of $f + \epsilon$, where $\epsilon \sim (0, C)$ is a random field with correlation C. Suppose that $C(x, x') = C(x - x')$

(That is, the covariance is stationary)

Then the eigenfunctions of the operator induced by C are the fourier basis

$$\int C(x - t)\phi_j(t)dt = c_j\phi_j(x)$$

where $\phi_j(x) = e^{-i2\pi x}$

Therefore, when we are getting the first few principal components, we really are getting the low frequency part of f

$$f_j = \int f\phi_j$$

Overview

A reasonable trichotomy of representation methods would be

- Probabilistic methods: Presumes to model the joint distribution of $p(X, Z)$, where Z are latent variables, and form the posterior $p(Z|X)$

  (A major example of this approach is graphical models)
- Auto-encoders: Creates a 'bottle-neck' in which a nonlinear function is sandwiched between a linear map and its transpose
- Manifold learning: Posits a lower-dimensional (but possibly nonlinear) manifold which the data live on or near and attempts to estimate it

Overview

Before covering these topics (and deep learning in particular), it will be helpful to cover two special cases

- PCA: Principal components can be phrased as an optimization problem over the Stiefel manifold of orthogonal matrices. This generalizes nicely to a version of deep learning
- Sparse coding: Leverages the intution that a good basis should allow for the features to be sparsely decomposable

PCA

PCA

REMINDER: Principal components is an (unsupervised) dimension reduction technique

It solves various equivalent optimization problems

(Maximize variance, minimize L_2 distortions, find closest subspace of a given rank,...)

At its core, we are finding linear combinations of the original (centered) covariates

$$Z_{ij} = \alpha_j^\top X_i$$

This is expressed algorithmically as the SVD $\mathbb{X} = UDV^\top$ and

$$Z_i = \mathbb{X} v_i = u_i d_i$$

PCA

If we want to find the first q principal components, the relevant optimization program is:

$$\min_{\mu,(\lambda_i),V_q} \sum_{i=1}^{n} ||X_i - \mu - V_q\lambda_i||^2$$

We can partially optimize for μ and (λ_i) to find

- $\hat{\mu} = \overline{X}$
- $\hat{\lambda}_i = V_q^\top(X_i - \hat{\mu})$

Now, we optimize

$$\min_{V \in \mathcal{S}_q} \sum_{i=1}^{n} \left|\left|(X_i - \hat{\mu}) - VV^\top(X_i - \hat{\mu})\right|\right|^2$$

where $\mathcal{S}_q$ is the Steifel manifold of rank-q orthogonal matrices

PCA

Principal components can be viewed as coming from all three representation learning paradigms

- Probabilistic methods: The leading eigenvectors of the covariance operator of the generative model
- Auto-encoders: It is a linear auto-encoder
- Manifold learning: Characterizing a lower-dimensional region in the covariate space where the data density is peaked

Images

- There are 575 total images
- Each image is 92×112 pixels and grey scale
- These images come from the Sheffield face database

  (See `http://www.face-rec.org/databases/` for this and other databases. See my rcode for how to read the images into R)

Faces

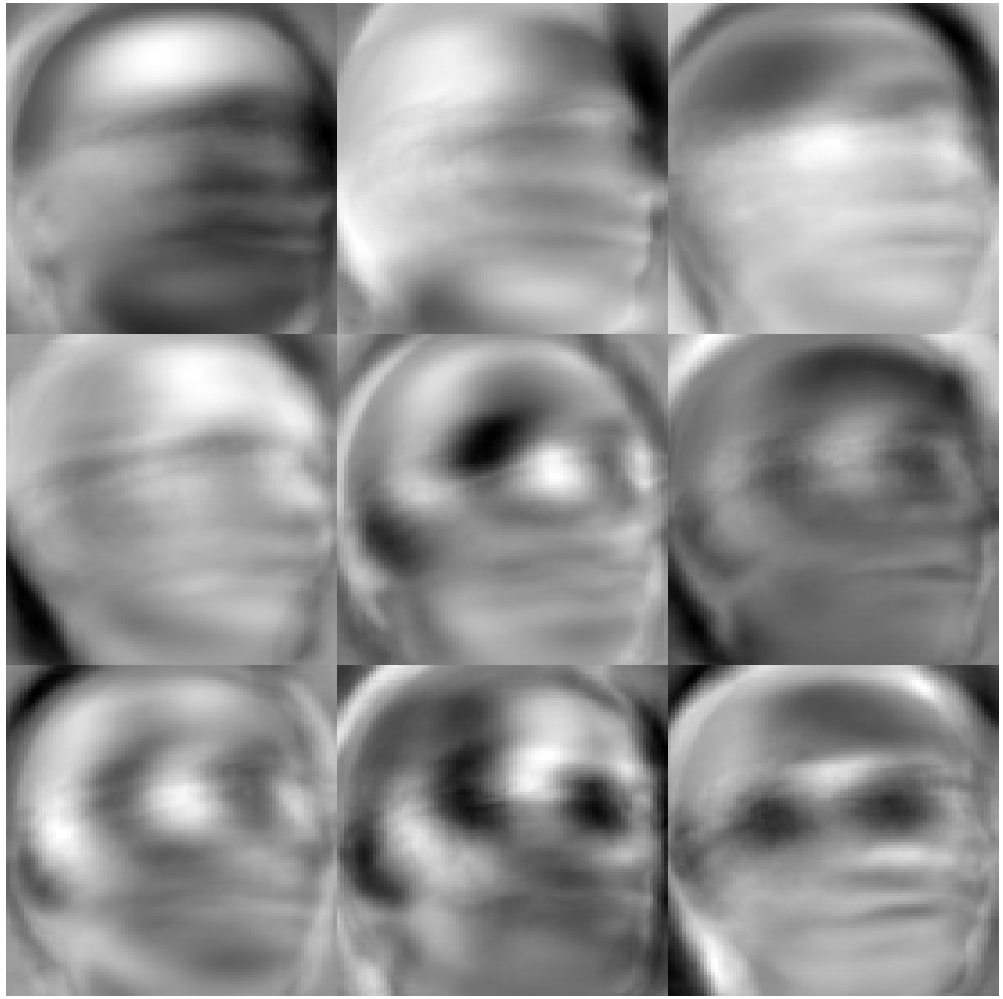
Faces

Regardless of how you formulate the optimization problem for PCA, it can be done in R by:

```
svd.out    = svd(scale(X,scale=F))
pc.basis   = svd.out$v
pc.scores = X %*% pc.basis
```

Let's apply this to the faces

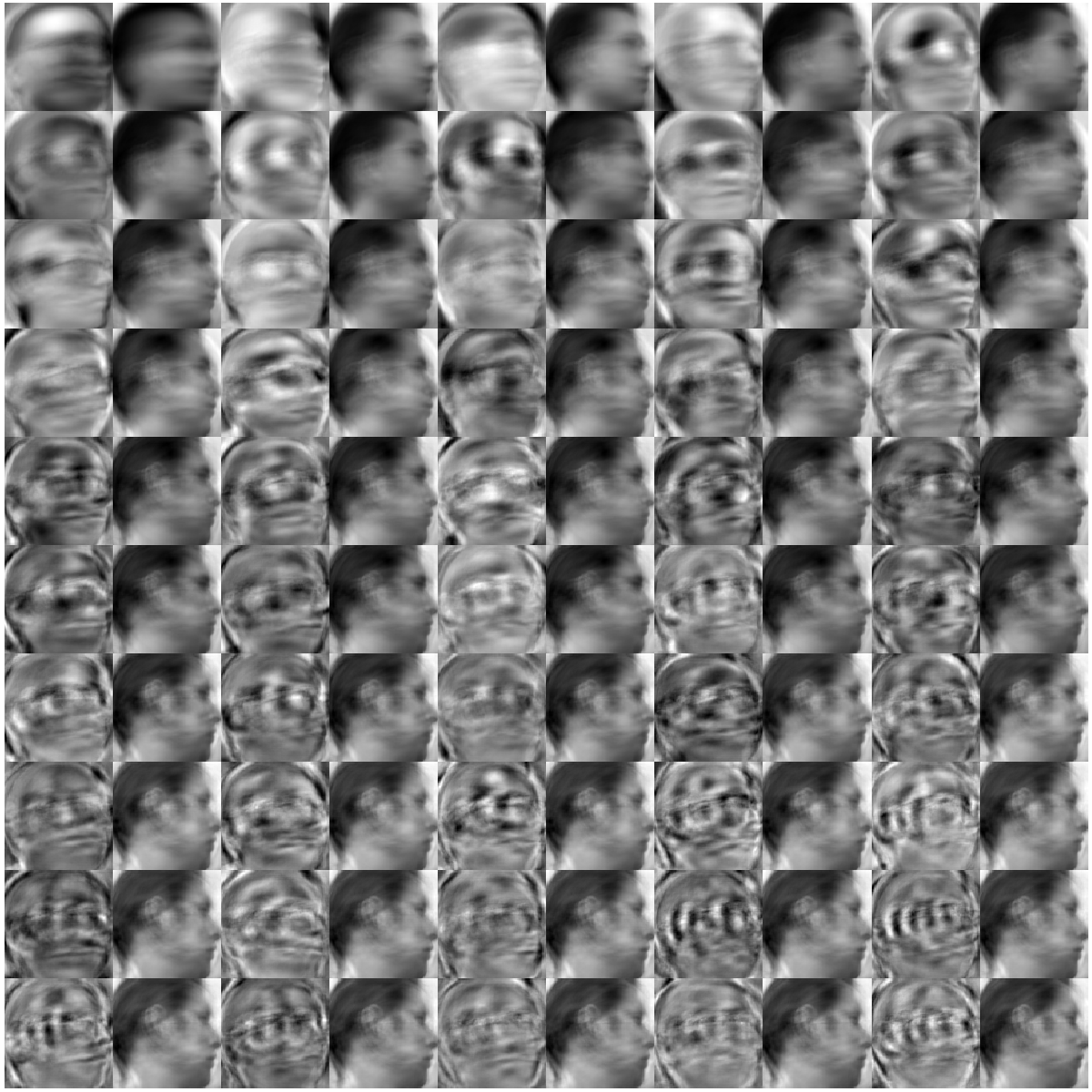
Faces: PC basis



Faces: PC projections

Varying levels of J: $\tilde{X} = \sum_{j=1}^{J} d_j u_j v_j^\top + \overline{X}$

Faces: PC projections and basis



Sparse coding

Sparse coding

From the same neurological background as neural nets, sparse coding was supposed to represent the workings of the mammalian visual-cortex

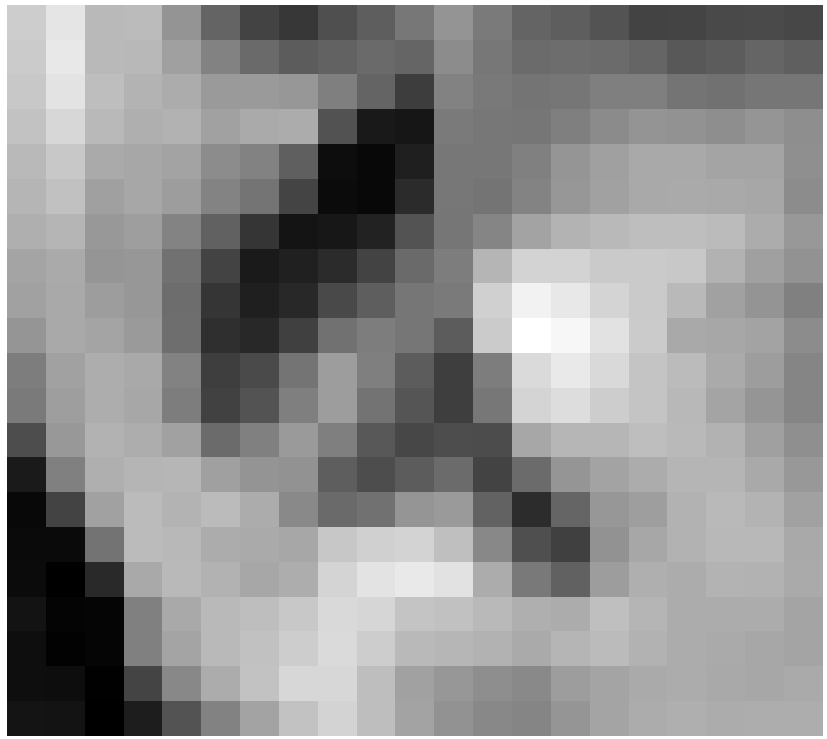
(See Olshausen, Field (1997) for the original (unreadable) paper and Marial et al. (2009) for a more SML take)

The idea is that we have adapted to certain types of images (such as forests) and can view them using only a few neurons

Mathematically: We possess (or have learned) a basis of neurons that permits certain types of images to be expressed sparsely

Sparse coding

We can represent the full image (left) on a computer using $92 \times 112 = 10304$ numbers



It is reasonable to assume that we can represent this image meaningfully using far less information

Sparse coding

The underlying presumption is that for $X \in \mathbb{R}^p$, suppose there is a dictionary Φ such that

$$X = \Phi\alpha$$

(Let's ignore noise for now)

Furthermore, presume that α is sparse in the sense of having few non-zero coefficients

Lastly, the dictionary $\Phi \in \mathbb{R}^{p \times K}$ is such that $K > p$ and composed of atoms ϕ_K

(The statement $K > p$ is known as overcomplete)

Sparse coding

Operationally, we take the idea of basis pursuit and convert it to generating a basis that can sparsely represent the signals we wish to decompose

Basis pursuit: This is effectively the lasso, but with the 'covariates' being some sort of basis. Let $\Phi = [\phi_1, \ldots, \phi_k]$ be such a matrix

(An example would be $\phi_1, \ldots, \phi_n$ being a wavelet basis, and $\phi_{n+1}, \ldots, \phi_{2n}$ being a Fourier basis)

Then, for a response Y (typically a signal such as an image)

$$\min_{\alpha} ||Y - \Phi\alpha||_2^2 + \lambda ||\alpha||_1$$

Sparse coding

Using the deviation-type inequalities, a tuning parameter λ can be set

$$\lambda_* = \sigma\sqrt{2\log(K)}$$

(Assuming the basis elements are ℓ_2 normalized)

Motivation: If Φ is orthogonal and $\tilde{Y} = \Phi^\top Y$ and

$$\hat{\alpha}_\lambda = \operatorname{sgn}(\tilde{Y})(|\tilde{Y}| - \lambda)_+ \quad \text{(Interpreted component-wise)}$$

Now, under certain assumptions about the noise, this is a normal means problem and $\hat{\alpha}_\lambda$ is the solution to the lasso Lagrangian

The series of papers by Donoho, Johnstone on wavelets imply that the λ_* choice is an optimal asymptotic MSE choice of the tuning parameter

(even when Φ is overcomplete)

Sparse coding

Sparse coding takes this idea and learns the basis Φ as well

Now, the problem is

$$\min_{\Phi, \alpha \in \mathbb{R}^{k \times n}} \sum_{i=1}^{n} (||X_i - \Phi \alpha_i||_2^2 + \lambda \, ||\alpha_i||)$$

$$\text{subject to } ||\Phi|| \leq c$$

A natural approach to this problem is to alternate between solving for α and Φ

(Both of these optimizations are constrained)

Sparse coding algorithm

A stochastic gradient descent approach tends to work well, where a random X_i is drawn, and the optimization is done with this example only

This alternation-based approach consists of

- Easy: Find α. This consists of doing a lasso-type solve. Usually this is done with a homotopy-type algorthm such as lars
- Hard: Find Φ. This depends on the nature of the constraint. There are varying methods for this

  (See Lee et al. (2007) for efficient algorithms based on the Lagrange dual. For online approaches that don't involve matrix inversion see Marial et al. (2009))

Sparse coding algorithm (online)

To solve for Φ, a classical, fast approach is a projected first-order stochastic gradient descent method

$$D_t = \Pi \left[D_{t-1} - \frac{\rho}{t} \nabla_\Phi \sum_{i=1}^{n} ||X_i - \Phi \alpha_i||_2^2 \right]$$

where

$$\nabla_\Phi \sum_{i=1}^{n} ||X_i - \Phi \alpha_i||_2^2 = \nabla_\Phi \left(\text{trace}(\Phi^\top \Phi A) - \text{trace}(\Phi^\top B) \right)$$

$$= 2\Phi A + B$$

where $A = \sum_{i=1}^{n} \alpha_i \alpha_i^\top$ and $B = \sum_{i=1}^{n} X_i \alpha_i^\top$

Sparse coding algorithm

Algorithm 1 Online dictionary learning.

Require: $\mathbf{x} \in \mathbb{R}^m \sim p(\mathbf{x})$ (random variable and an algorithm to draw i.i.d samples of p), $\lambda \in \mathbb{R}$ (regularization parameter), $\mathbf{D}_0 \in \mathbb{R}^{m \times k}$ (initial dictionary), T (number of iterations).

1: $\mathbf{A}_0 \leftarrow 0$, $\mathbf{B}_0 \leftarrow 0$ (reset the "past" information).
2: **for** $t = 1$ to T **do**
3: Draw $\mathbf{x}_t$ from $p(\mathbf{x})$.
4: Sparse coding: compute using LARS

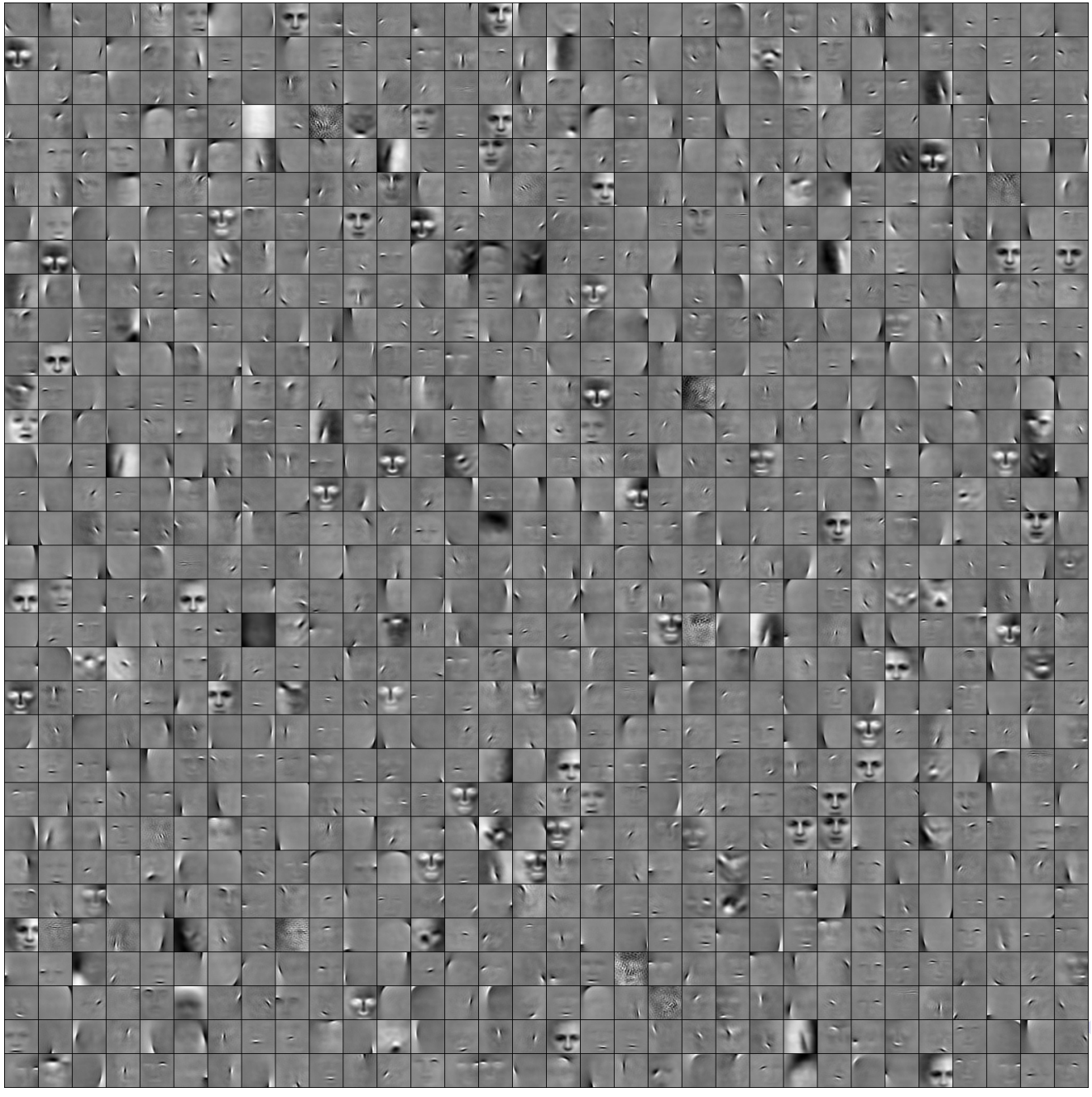
$$\boldsymbol{\alpha}_t \triangleq \arg\min_{\boldsymbol{\alpha} \in \mathbb{R}^k} \frac{1}{2}||\mathbf{x}_t - \mathbf{D}_{t-1}\boldsymbol{\alpha}||_2^2 + \lambda||\boldsymbol{\alpha}||_1. \quad (8)$$

5: $\mathbf{A}_t \leftarrow \mathbf{A}_{t-1} + \boldsymbol{\alpha}_t \boldsymbol{\alpha}_t^T$.
6: $\mathbf{B}_t \leftarrow \mathbf{B}_{t-1} + \mathbf{x}_t \boldsymbol{\alpha}_t^T$.
7: Compute $\mathbf{D}_t$ using Algorithm 2, with $\mathbf{D}_{t-1}$ as warm restart, so that

$$\begin{aligned} \mathbf{D}_t &\triangleq \arg\min_{\mathbf{D} \in \mathcal{C}} \frac{1}{t} \sum_{i=1}^{t} \frac{1}{2}||\mathbf{x}_i - \mathbf{D}\boldsymbol{\alpha}_i||_2^2 + \lambda||\boldsymbol{\alpha}_i||_1, \\ &= \arg\min_{\mathbf{D} \in \mathcal{C}} \frac{1}{t}\left(\frac{1}{2}\operatorname{Tr}(\mathbf{D}^T\mathbf{D}\mathbf{A}_t) - \operatorname{Tr}(\mathbf{D}^T\mathbf{B}_t)\right). \end{aligned} \quad (9)$$

8: **end for**
9: **Return** $\mathbf{D}_T$ (learned dictionary).

Sparse coding: Images



Sparse coding: Images

Some comments:

- See http://www.cs.tau.ac.il/~wolf/ytfaces/ for a database of unaligned faces
- I got this panel of faces from http://charles.cadieu.us/?p=184.

  (See the website and Olshausen et al. (2009) for details)

Deep learning

Deep learning: Overview[8]

Neural networks are models for supervised learning

Linear combinations of features are fed through nonlinear functions repeatedly

At the top layer, the resulting latent factor is fed into a linear/logistic regression

[8] These notes are largely from a conversation with Rob Tibshirani. The ideas contained herein are partially his, and will appear in a future book 'L1 methods and the Lasso'

Deep learning: Overview

As far as I can tell, deep learning is a new way of fitting neural nets

The central idea is referred to as greedy layerwise unsupervised pre-training (Terminology appeared in Bengio et al. (2007))

Here, we wish to learn a hierarchy of features one level at a time, using

1. unsupervised feature learning to learn a new transformation at each level
2. which gets composed with the previously learned transformations

Essentially, each iteration of unsupervised feature learning adds one layer of weights to a deep neural network

The top layer is used to initialize a (supervised) neural network

Deep learning: Overview

Traditionally, a neural net is fit to all labelled data in one operation, with weights randomly chosen near zero

Due to the nonconvexity of the objective function, the final solution can get 'caught' in poor local minima

Deep learning seeks to find a good starting value, while allowing for:

- ...modeling the joint distribution of the covariates separately
- ...use of unlabeled data (included the test covariates)

Auto-encoders

In neural networks the idea of a auto encoder generalizes the ideas of PCA and sparse coding by

- Using multiple hidden layers, leading to a hierarchy of dictionaries

  (As PCA is linear, composing multiple layers adds no generality. Sparse coding provides only 1 layer between covariates and the representation)

- applying the encoding models to local patches of an image, commonly with weight sharing where constraint weights are enforced to be equal across an image

  (This is the so-called convolutional neural network framework)

Auto-encoders

An auto-encoder is comprised of (LeCun (1987); Hinton, Zemel (1994)):

- Feature-extracting function: This function $h : \mathbb{R}^p \to \mathbb{R}^K$ maps the covariates to a new representation and is also known as the encoder
- Reconstruction function: This function[9] $h^{-1} : \mathbb{R}^K \to \mathbb{R}^p$ is also known as the decoder and it maps the representation back into the original space

Goal: Optimize any free parameters in the encoder/decoder pair that minimizes reconstruction error

[9] I've labeled this function h^{-1} to be suggestive, but I don't mean that $h^{-1}(h(x)) = x$

Auto-encoders

Of course this means some sort of implicit or explicit constraint need be imposed to not learn the identity function

This comes about via a combination of

- ... Regularization

  (Usually called a regularized auto-encoder)

- ... Dimensional constraint

  (Usually called a classical auto-encoder)

All flavors essentially reduce to solving the following optimization problem (perhaps with constraints)

$$\min \sum_{i=1}^{n} \ell(X_i, h^{-1}h(X_i))$$

Classic auto-encoder

As auto-encoders were first presented in the context on neural networks, they tend to have the following linear[10] form:

Let $W \in \mathbb{R}^{p \times K}$ (with $K < p$) be a matrix of weights

Each linear combination of an input vector X is fed through a nonlinear function σ, creating

$$h(X) = \sigma(W^\top X) \in \mathbb{R}^K$$

The output layer is then modeled as a linear combination of these inputs[11]

$$h^{-1}(h(X)) = Wh(X) = W\sigma(W^\top X) \in \mathbb{R}^p$$

[10] Really, it is affine with the inclusion of a bias term

[11] There is no restriction that the same matrix to be used in h and h^{-1}. Keeping them the same is known as weight-tying

Deep learning

Given inputs $X_1, \ldots, X_n$, the weight matrix W is estimated by solving the (non convex) optimization problem:

$$\min_{W \in \mathbb{R}^{p \times K}} \sum_{i=1}^{n} ||X_i - Wh(X_i)||^2$$

If $\sigma(X) \equiv X$, then $h(X) = W^\top X$ and we've recovered the PCA program

(In the sense that we've recovered the same subspace)

Deep learning

The framework is determined by the relative sizes of K and p

- If $K < p$, the rank constraint provides a bottleneck in the network that forces the learning of structure

  (e.g. PCA)
- If $K > p$, the representation is overcomplete and some regularization is needed

  (e.g. sparse coding)

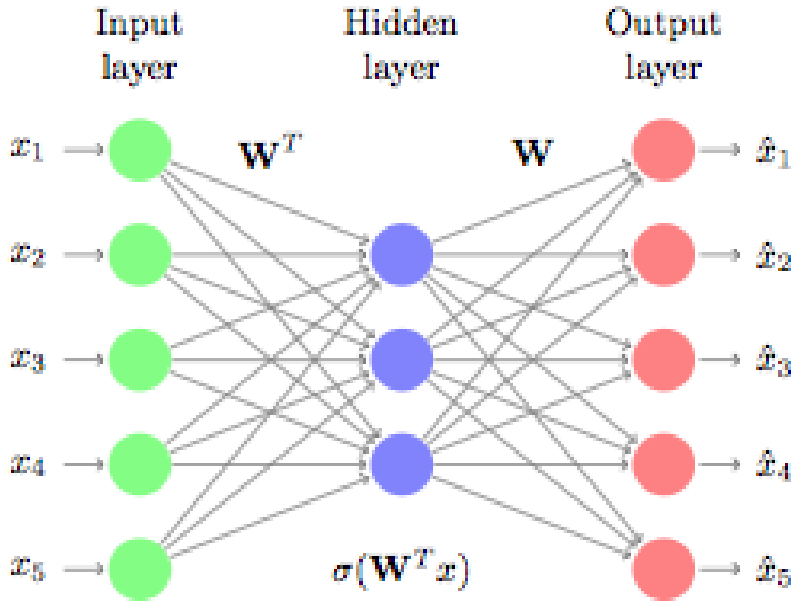
  Regularization comes about in several ways
  - Adding a regularization term on the parameters to the objective function
  - Corrupting the inputs before auto-encoding and comparing to uncorrupted inputs

    (This is known as a denoising auto-encoder)
  - Adding a regularization term on the Jacobian of the encoder to the objective function

    (This is known as a contractive auto-encoder)

Deep learning schematic

A rank constrained deep learning implementation might look like:



Deep learning

Modern deep learning generalize the previous definition in several ways

(See Le, Ranzato, Monga, Devin, Chen, Dean, Ng (2012) for details)

- They use multiple hidden layers, leading to a hierarchy of dictionaries
- Include nonlinearities that can be computed faster (such as $\sigma(x) = x_+$
- The encoding is applied to local patches of images (or signals) and these patches might be forced to have the same weights, imposing weight-sharing or a convolutional structure

Deep learning

The following is one of the most state-of-the-art implementation of deep learning I'm aware of

(Le, Ranzato, Monga, Devin, Chen, Dean, Ng (2012))

It has about 1 billion trainable parameters and uses advanced parallelism to make computation feasible

It also uses a decoupled encoder-decoder pair, plus regularization and a linear activation

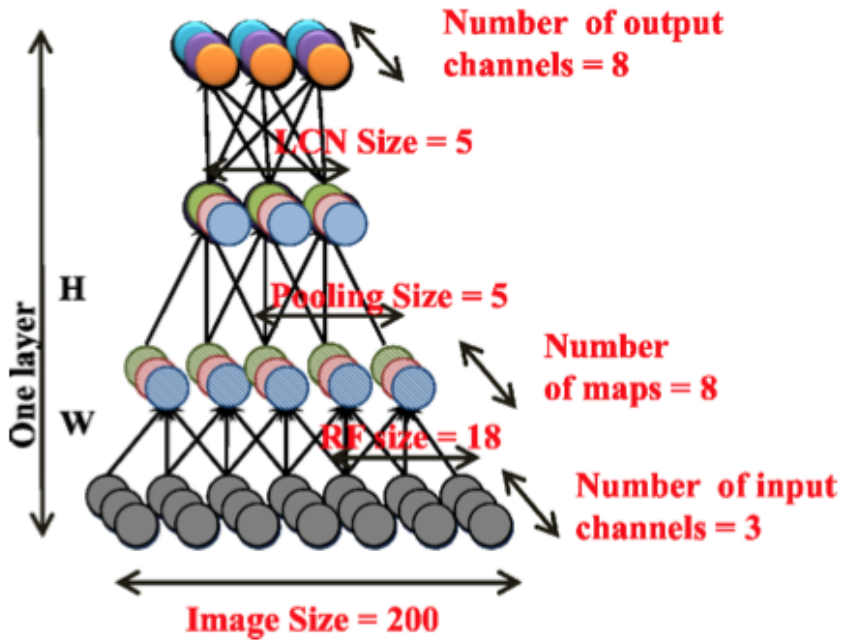
$$\min_{W_1, W_2} \sum_{i=1}^{n} \left(\left|\left| W_2 W_1^\top X_i - X_i \right|\right|_2^2 + \lambda \sum_{k=1}^{K} \sqrt{h_k (W_1^\top X_i)^2} \right)$$

where h_k are the pooling weights

(These are usually not optimized over and set to uniform weighting)

Deep learning schematic

A regularized deep learning implementation might look like:



Deep learning

This has three important ingredients

- Local receptive fields: Grab each patch in input image and transform into feature in second layer

  (If convolutional or weight-sharing these maps will all have the same weights)

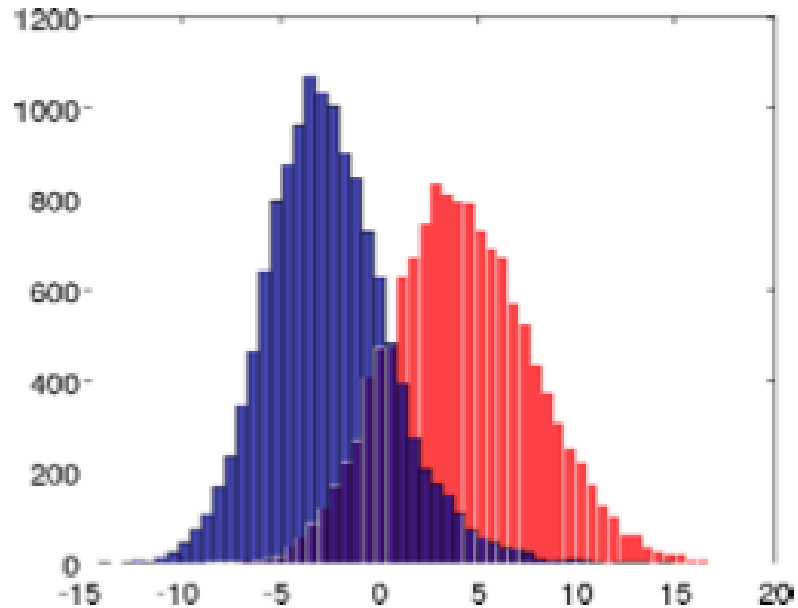
- Pooling: To achieve invariance to local structures, take the $\sqrt{(\cdot)^2}$ of its inputs
- Local contrast normalization: This locally standardizes each neuron and is usually interpreted as measure of fitness

  (It is motivated by computational neuroscience models (e.g. Pinto, Cox, DiCarlo (2008) and has been shown empirically to improve results (Jarrett et al (2009)))

Deep learning results

If we look at every neuron (that is, hidden unit) in the network and take the output for a given body of test images

Maximize the classification rate of taking the $\mathrm{sign}(\cdot)$, they find:



Deep learning results

The test images with maximum activation of that optimal neuron

Deep learning results

Finding the pixel wise maximizing input: