Neural Networks and Deep Learning

-Statistical Machine Learning-

Lecturer: Darren Homrighausen, PhD

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)

(Chapter 11 in ESL is a good introductory reference for neural networks)

Background

Background

Neural networks have come about in 3 "waves" of research

The first was an attempt to model the mechanics of the human brain

Through psychological and anatomical experimentation, it appeared the brain worked by

- taking atomic units known as neurons, which can either be "on" or "off"
- putting them in networks with each other, where the signal is given by which neurons are "on" at a given time

Crucially, a neuron itself interprets the status of other neurons

Background

After the development of parallel, distributed computation in the 1980s, the artificial intelligence view was diminished

Neural networks became popular machine learning approaches

Among the growing popularity of SVMs and boosting/bagging in the late 1990s, neural networks again fell out of favor

This was due to many of the problems we'll discuss (non convexity being the main one)

Background

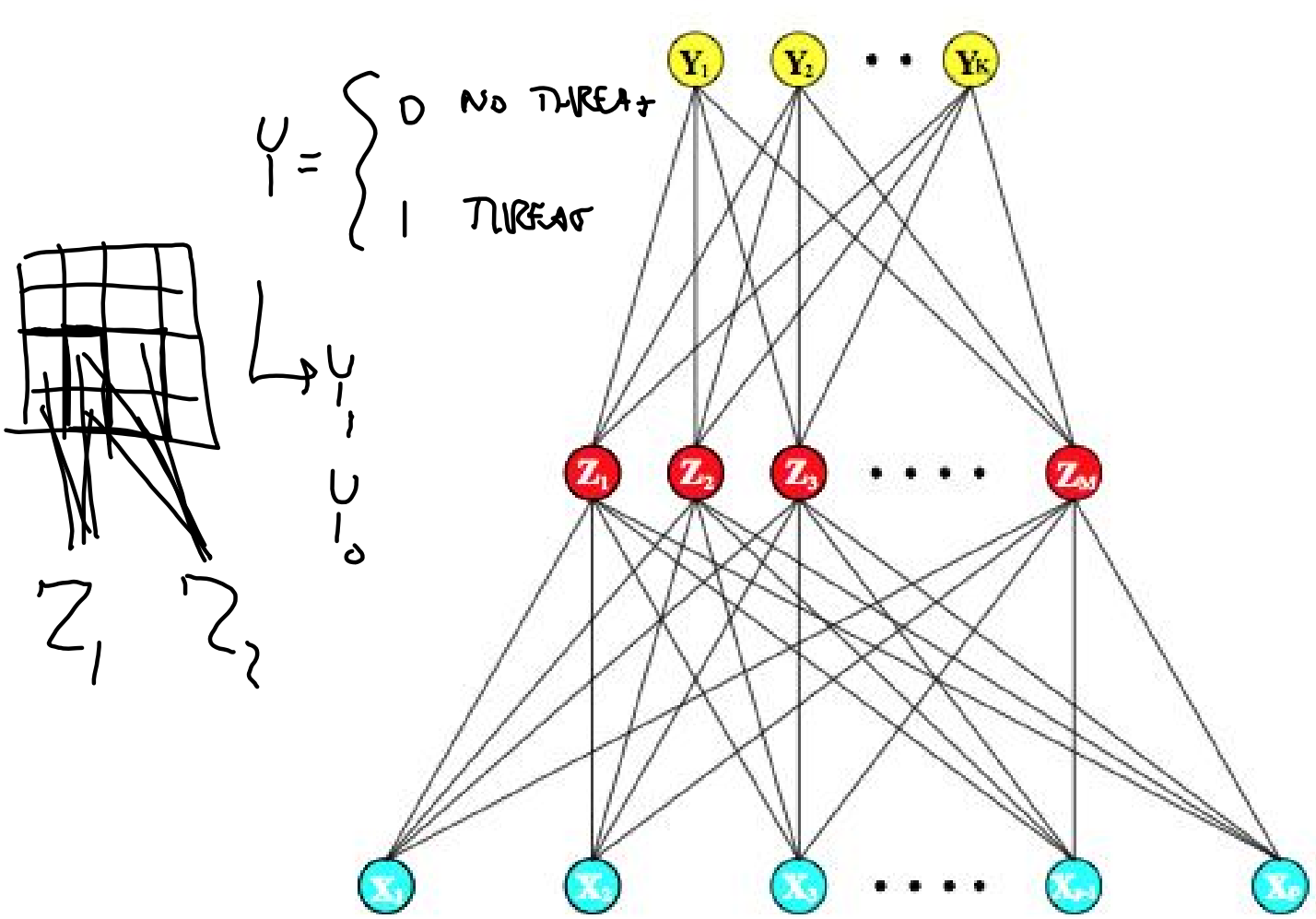
In the mid 2000's, new approaches for initializing neural networks became available

These approaches are collectively known as deep learning

Together, some state-of-the-art performance on various classification tasks have been accomplished via neural networks

High level overview

High level overview



$Y \mid X, \beta = N(X^T\beta, \sigma^2)$

$\beta \sim N(0, \Sigma)$

$\Sigma \sim IP$

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

$$\mathbb{E}Y|X = f_*(X) \stackrel{?}{=} \beta^T X$$

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

- A fixed basis, $(\phi_k)_{k=1}^{\infty}$
- A tuning parameter k

Nonparametric regression

We follow this prescription:

1. Write[1]

$$f_*(X) = \sum_{k=1}^{\infty} \beta_k \phi_k(X) \quad \text{“}=\text{”} \quad \sum \beta_k \chi_k$$

where $\beta_k = \langle f_*, \phi_k \rangle$

2. Truncate this expansion[2] at K

f_* HAS ∞ DERIVATIVES

$|\beta_k| \lesssim k^{-\omega}$

$$f_*^K(X) = \sum_{k=1}^{K} \beta_k \phi_k(X)$$

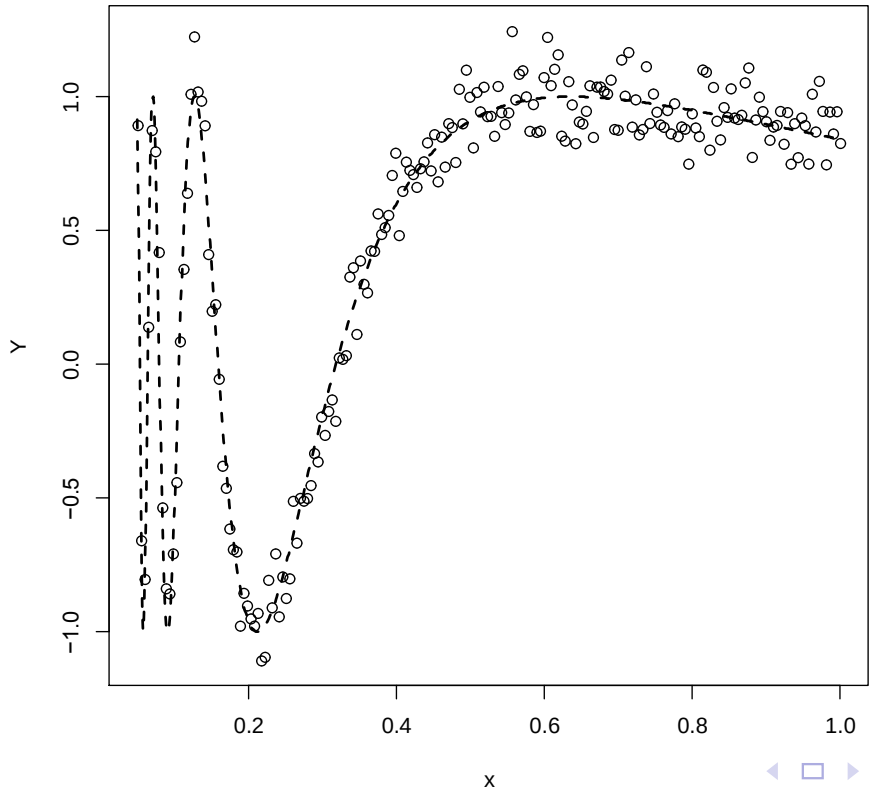
3. Estimate β_k with least squares

[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 k are more rough $\Rightarrow$ this is a smoothness assumption

Nonparametric regression: Example

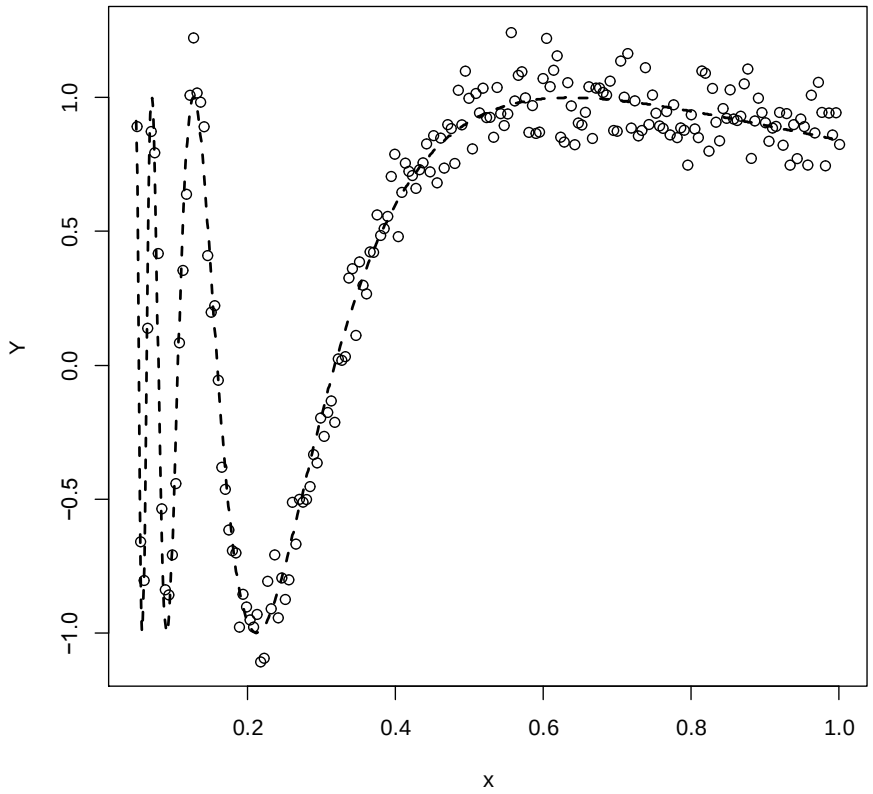
```
x = seq(.05,1,length=200)
Y = sin(1/x) + rnorm(100,0,.1)
plot(x,Y)
xTest = seq(.05,1,length=1000)
lines(xTest,sin(1/xTest),col='black',lwd=2,lty=2)
```

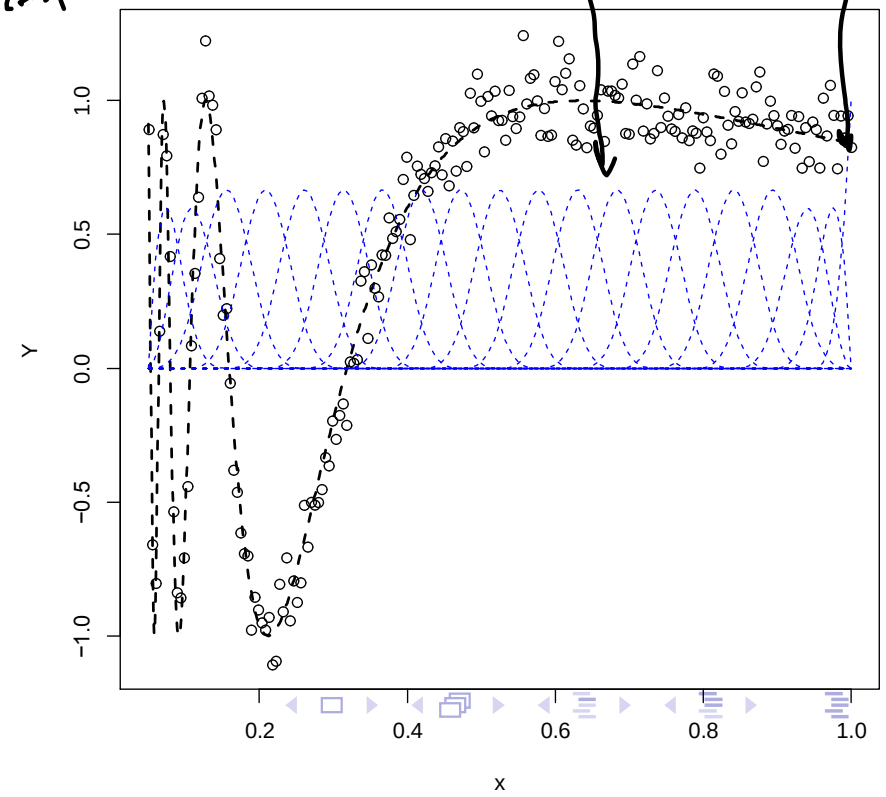


"Doppler"

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')
```



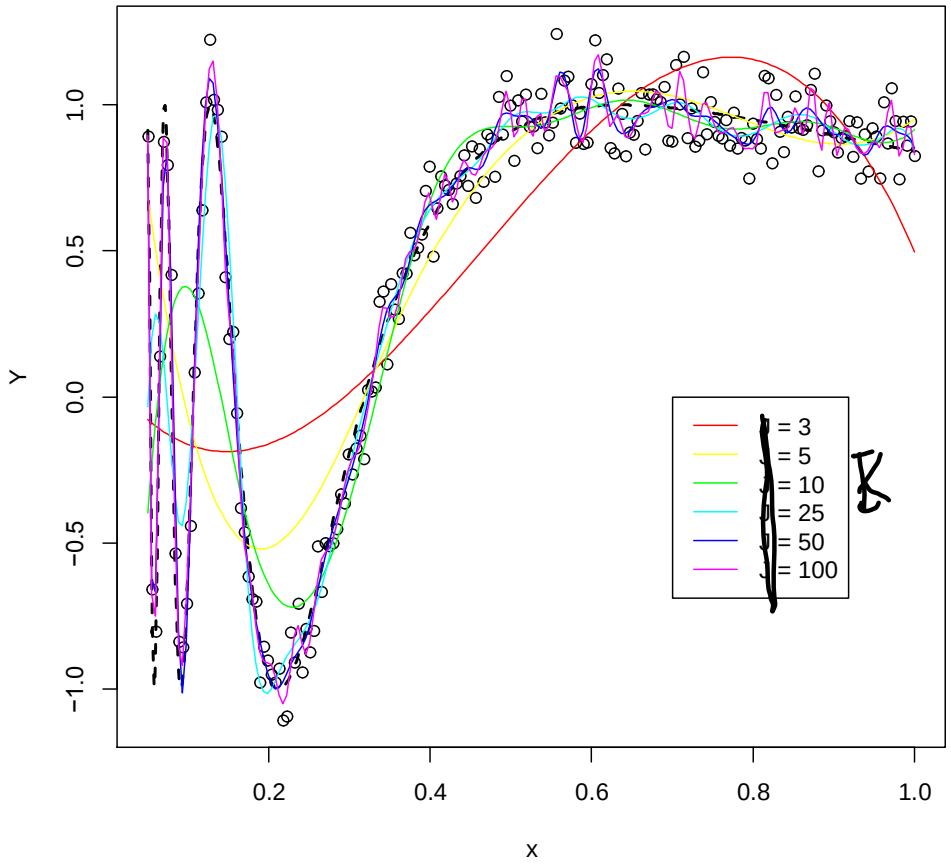


Nonparametric regression: Example

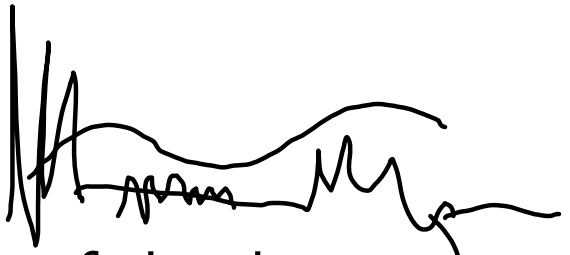
```
require(splines)
X    = bs(x,df=K)
Yhat = predict(lm(Y~.,data=X))
```

BIAS

$$\sum_{k=K+1}^{\infty} |\beta_k|^2$$



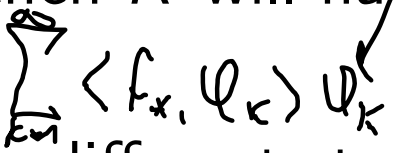
Nonparametric regression



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 K 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

Let $\mu(X) = \mathbb{E}Y|X$

Write L as the link function

A basic[3] neural network can be phrased

$$L(\mu(X)) = \beta_0 + \sum_{k=1}^{K} \beta_k \sigma(\alpha_{k0} + \alpha_k^\top X)$$

Compare: A nonparametric GLM would have the form

$$L(\mu(X)) = \beta_0 + \sum_{k=1}^{K} \beta_k \phi_k(X)$$

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

Neural networks: Definitions

$$L(\mu(X)) = \beta_0 + \sum_{k=1}^{K} \beta_k \sigma(\alpha_{k0} + \alpha_k^\top X)$$

The main components are

- The derived features $Z_k = \sigma(\alpha_{k0} + \alpha_k^\top X)$ and are called the hidden units
  - The function σ is called the activation function and is very often $\sigma(u) = (1 + e^{-u})^{-1}$

    (This particular $\sigma(u)$ is known as the sigmoid function)
  - The parameters $\beta_0, \beta_k, \alpha_{k0}, \alpha_k$ are estimated from the data.
- The number of hidden units K is a tuning parameter

High level overview

Example: If $L(\mu) = \mu$, then we are doing regression:

$$\mu(X) = \beta_0 + \sum_{k=1}^{K} \beta_k \sigma \left(\alpha_{k0} + \sum_{j=1}^{p} \alpha_{kj} x_j \right)$$

but in a transformed space

Two observations:

- The σ function generates a feature map
- If $\sigma(u) = u$, then neural networks reduce to classic least squares

Let's discuss each of these..

Observation 1: Feature map

We start with p covariates

We generate K features

Example: GLMs with a feature transformation

$$\Phi(X) = (1, x_1, x_2, \ldots, x_p, x_1^2, x_2^2, \ldots, x_p^2, x_1x_2, \ldots, x_{p-1}x_p) \in \mathbb{R}^K$$
$$= (\phi_1(X), \ldots, \phi_K(X))$$

Before feature map:

$$L(\mu(X)) = \beta_0 + \sum_{j=1}^{p} \beta_j x_j$$

After feature map:

$$L(\mu(X)) = \beta^\top \Phi(X) = \sum_{k=1}^{K} \beta_k \phi_k(X)$$

$\phi_k(X) \rightsquigarrow Z_k$

Observation 1: Feature map

For neural networks write:

$$Z_k = \sigma\left(\alpha_{k0} + \sum_{j=1}^{p} \alpha_{kj} x_j\right) = \sigma\left(\alpha_{k0} + \alpha_k^\top X\right)$$

Then we have

$$\Phi(X) = (1, Z_1, \ldots, Z_K)^\top \in \mathbb{R}^{K+1}$$

and

$$\mu(X) = \beta^\top \Phi(X) = \beta_0 + \sum_{k=1}^{K} \beta_k \sigma\left(\alpha_{k0} + \sum_{j=1}^{p} \alpha_{kj} x_j\right)$$

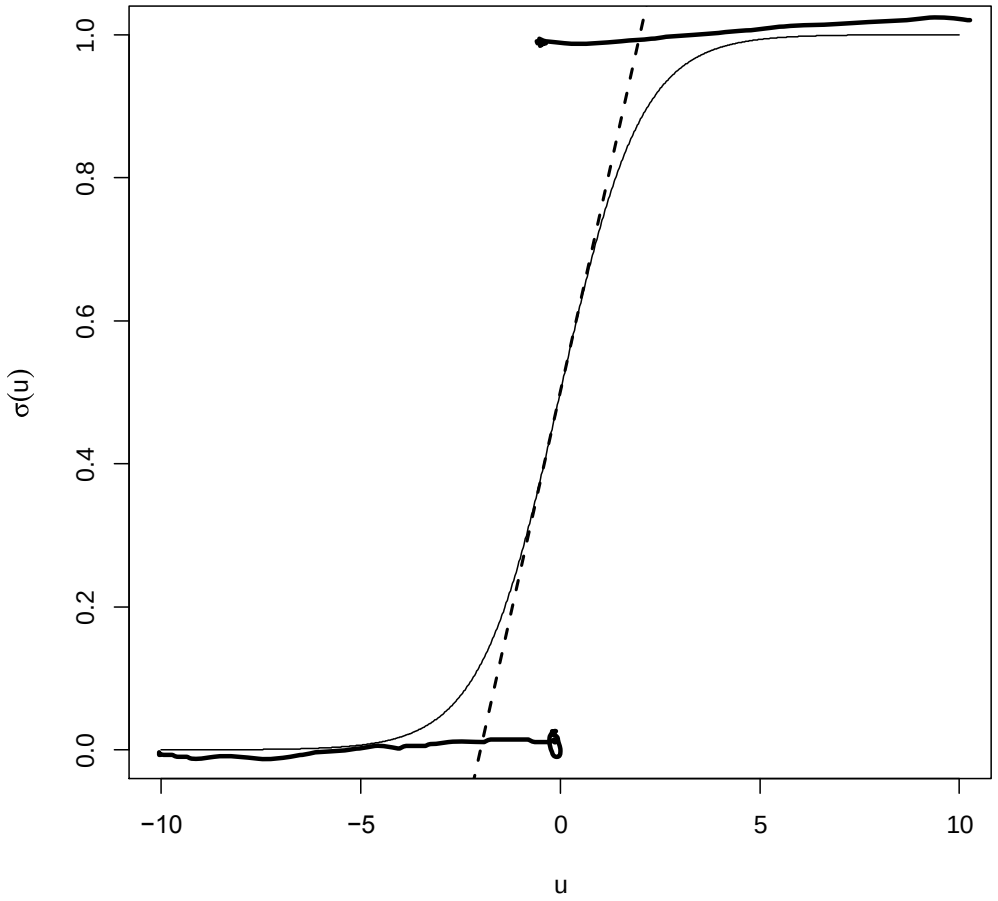
Observation 2: Activation function

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

$$
\begin{aligned}
L(\mu(X)) &= \beta_0 + \sum_{k=1}^{K} \beta_k \sigma(\alpha_{k0} + \alpha_k^\top X) \\
&= \beta_0 + \sum_{k=1}^{K} \beta_k (\alpha_{k0} + \alpha_k^\top X) \\
&= \beta_0 + \sum_{k=1}^{K} \beta_k \alpha_{k0} + \sum_{k=1}^{K} \beta_k \alpha_k^\top X \\
&= \gamma_0 + \gamma^\top X \\
&= \gamma_0 + \sum_{j=1}^{p} \gamma_j^\top x_j
\end{aligned}
$$

Observation 2: Activation function

Plot of sigmoid activation function



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

Hierarchical model

A neural network can be phrased as a hierarchical model

$$Z_k = \sigma(\alpha_{k0} + \alpha_k^\top X) \quad (k=1,\dots K)$$
$$W_g = \beta_{g0} + \beta_g^\top Z \quad (g=1,\dots G)$$
$$\mu_g(X) = L^{-1}(W_g)$$

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

- Regression: The link function is $L(u) = u$

  (here, $G = 1$)
- Classification: With G classes, we are modeling $\pi_g = \mathbb{P}(Y = g|X)$ and $L = \text{logit}$:

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

(This is called the softmax function for historical reasons)

Training neural networks

Neural networks have many (MANY) unknown parameters

(They are usually called weights in this context)

These are

- α_{k0}, α_k for $k = 1, \ldots, K$ (total of $K(p+1)$ parameters)
- β_{g0}, β_g for $g = 1, \ldots, G$ (total of $G(K+1)$ parameters)

Total parameters: $\asymp Kp + GK = K(p+G)$

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

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

  - 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

Training neural networks

We'll need some derivatives to implement the gradient descent

$$\mu(X) = \beta_0 + \sum_{k=1}^{K} \beta_k \sigma \left(\alpha_{k0} + \sum_{j=1}^{p} \alpha_{kj} x_j \right)$$

Derivatives:

$$\frac{\partial \mu}{\partial \beta_k} = \sigma(\alpha_{k0} + \alpha_k^\top X) = Z_k$$

$$\frac{\partial \mu}{\partial \alpha_{kj}} = \beta_k \sigma'(\alpha_{k0} + \alpha_k^\top X) x_j$$

Neural networks: Back-propagation

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

Then

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

$$\frac{\partial \hat{R}_i}{\partial \alpha_{kj}} = -2(Y_i - \hat{Y}_i)\beta_k \sigma'(\alpha_{k0} + \alpha_k^\top X_i)X_{ij}$$

Given these derivatives, a gradient descent update can be found

$$\hat{\beta}_k^{t+1} = \hat{\beta}_k^t - \gamma_t \sum_{i=1}^n \frac{\partial \hat{R}_i}{\partial \beta_k}\bigg|_{\hat{\beta}_k^t}$$

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

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

Neural networks: Back-propagation

Returning to

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

$$\frac{\partial \hat{R}_i}{\partial \alpha_{kj}} = -2(Y_i - \hat{Y}_i)\beta_k \sigma'(\alpha_{k0} + \alpha_k^\top X_i)X_{ij} \qquad = b_{ki} X_{ij}$$

Direct substitution of a_i into b_{ki} gives

$$b_{ki} = a_i \beta_k \sigma'(\alpha_{k0} + \alpha_k^\top X_i)$$

These are the back-propagation equations

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 $\hat{R}_i$, 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

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
- Modified globally convergent version

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, with some combination of:

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

Regularizing neural networks

Explicit regularization comes in a couple of flavors

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

$$\rho(\alpha, \beta) = \sum \beta^2 + \sum \alpha^2$$

- Weight elimination: This encourages more shrinking of small weights

$$\rho(\alpha, \beta) = \sum \frac{\beta^2}{1 + \beta^2} + \sum \frac{\alpha^2}{1 + \alpha^2}$$

Note: In either case, we now solve:

$$\min \hat{R} + \lambda \rho(\alpha, \beta)$$

This can be done efficiently by augmenting the gradient descent derivatives

Common pitfalls

There are three 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 (K): It is generally better to have too many hidden units than too few (regularization can eliminate some).

  (Later, we will see this can include 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 linear 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)