Neural Networks and Deep Learning

-Introduction to Data Science-

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

We will see that neural networks are another way of fitting the nonparametric model

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

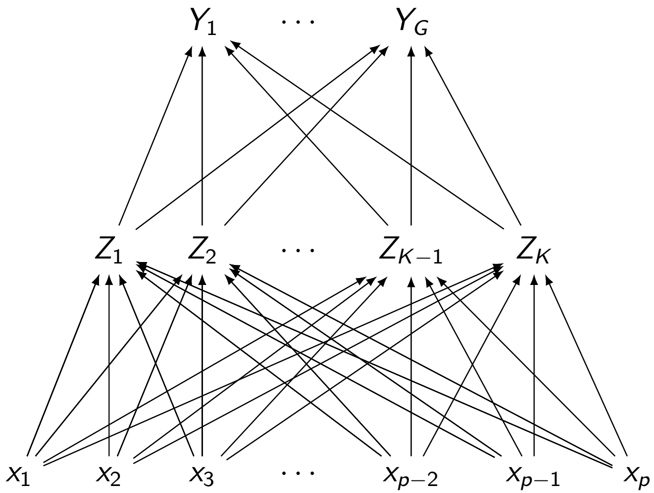
At the top layer, the resulting latent factors are fed into an algorithm for predictions

(Most commonly via least squares or logistic regression)

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

High level overview

Hierarchical view



Y_1 $\cdots$ Y_G
Z_1 Z_2 $\cdots$ Z_{K-1} Z_K
x_1 x_2 x_3 $\cdots$ x_{p-2} x_{p-1} x_p

Recall: This is a (single hidden layer) neural network. Note the similarity to latent factor models

High level overview

Owing to the original anatomical original of neural networks, a key component is the activation function

For historical reasons, it is commonly notated as a function σ
(Not to be confused with the usual notation for standard deviation σ)

For now, think of the activation function as the amount of signal required before a neuron turns “on”

We will look into the activation function more closely later

High level overview

Let $\mu(X) = \mathbb{E}Y|X$ be the conditional mean of the supervisor given the feature vector

Reminder: GLMs have two components:

- A systematic component: $f(X) = X^\top \beta$
- A link function: L

The link function L relates the mean $\mu(X)$ to the systematic component $f(X)$

Example: If $Y|X \sim$Bernoulli$(\pi(X))$

(Note that if $Y \in \{0,1\}$, then $\mu(X) = \pi(X) = \mathbb{P}(Y = 1|X)$)

Then $L(\mu(X)) = \log(\mu(X)/(1 - \mu(X)))$ is logistic regression

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

High level overview

GLMs

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

(Example: $L(\mu) = \log(\mu/(1-\mu))$ is logistic regression)

Compare to: A single layer neural network is:

$$\begin{aligned} L(\mu(X)) &= \beta_0 + \sum_{k=1}^{K} \beta_k \phi_k(X) \\ &= \beta_0 + \sum_{k=1}^{K} \beta_k Z_k \\ &= \beta_0 + \sum_{k=1}^{K} \beta_k \sigma\left(\alpha_{k0} + \sum_{j=1}^{p} \alpha_{kj} x_j\right) \end{aligned}$$

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
  - 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

IMPORTANT: If we define $\phi_k(X) = \sigma\left(\alpha_{k0} + \sum_{j=1}^{p} \alpha_{kj} x_j\right)$, then we see that, like boosting, neural networks estimate both the coefficients and basis functions

(However, unlike boosting, neural networks estimate the coefficients and basis functions in a different manner and hence do not behave similarly)

High level overview

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

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

but in a transformed space

Two observations:

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

Let's discuss each of these..

Observation 1: Feature map

We start with p features

We generate K features transformations

Example: GLMs with a feature transformation

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

Before feature map:

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

After feature map:

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

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

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

and

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

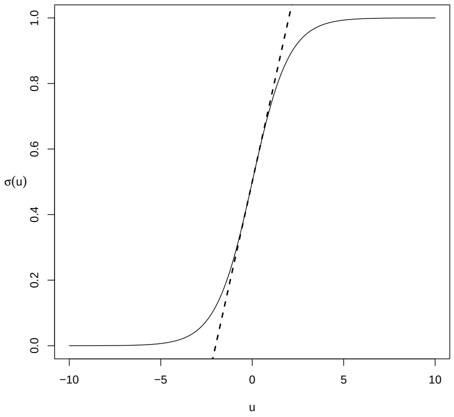
Observation 2: Activation function

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

$$\begin{aligned}
L(\mu(X)) &= \beta_0 + \sum_{k=1}^{K} \beta_k \sigma(\alpha_{k0} + \alpha_k^\top X) \\
&= \beta_0 + \sum_{k=1}^{K} \beta_k (\alpha_{k0} + \alpha_k^\top X) \\
&= \beta_0 + \sum_{k=1}^{K} \beta_k \alpha_{k0} + \sum_{k=1}^{K} \beta_k \alpha_k^\top X \\
&= \tilde{\beta}_0 + \tilde{\beta}^\top X \\
&= \tilde{\beta}_0 + \sum_{j=1}^{p} \tilde{\beta}_j x_j
\end{aligned}$$

Observation 2: Activation function

A classic activation function is the sigmoid $\sigma(u) = (1 + e^{-u})^{-1}$



If we look at a plot of the sigmoid function, it is quite linear for $u \approx 0$ or $|u|$ large, but has nonlinear behavior at intermediate values of u

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

One-hot encoding

There are many ways to convey information in a binary system

(Binary: numbering system with only two possible values, usually chosen as 0 and 1)

One way is via one-hot encoding in which only one entry (known as a bit) is allowed to be 1 and all others must be a 0.

(This system can only send as much information as there are bits, however)

In neural networks for classification it is standard to...

... encode the supervisor as a $n \times G$ matrix of $0/1$'s

instead of...

... a $n \times 1$ vector of $1, 2, \ldots, G$'s

(Or whatever the relevant levels of Y are)

So, with one-hot encoding, $Y \in \mathbb{R}^{n \times G}$ is a matrix, with each row having $G - 1$ "zeros" and 1 "one"

Training neural networks

The most common loss functions are

- Regression: Squared error

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

- 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 $Y_i \in \mathbb{R}^G$

    (This is one-hot encoding)
  - 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

For neural networks, 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

Gradient descent:

$$\hat{\beta}_k^{t+1} = \hat{\beta}_k^t - \lambda_t \sum_{i=1}^n \left. \frac{\partial \ell(f(X_i), Y_i)}{\partial \beta_k} \right|_{\hat{\beta}_k^t}$$

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

Training neural networks

We'll need some derivatives to implement the gradient descent

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

Derivatives:

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

(In other words, the derivative w.r.t. β_k is just the hidden unit)

Neural networks: Back-propagation

For squared error, let $\ell(f(X_i), Y_i) = (Y_i - f(X_i))^2$

(Hence, the training error is $\hat{R}(f) = \frac{1}{n}\sum_{i=1}^{n}\ell(f(X_i), Y_i)$)

Then

$$\frac{\partial \ell(f(X_i), Y_i)}{\partial \beta_k} = -2(Y_i - f(X_i))Z_{ik}$$

$$\frac{\partial \ell(f(X_i), Y_i)}{\partial \alpha_{kj}} = -2(Y_i - f(X_i))\beta_k \sigma'(\alpha_{k0} + \alpha_k^\top X_i)X_{ij}$$

Given these derivatives, a gradient descent update is:

$$\hat{\beta}_k^{t+1} = \hat{\beta}_k^t - \lambda_t \sum_{i=1}^{n} \frac{\partial \ell(f(X_i), Y_i)}{\partial \beta_k}\bigg|_{\hat{\beta}_k^t}$$

$$\hat{\alpha}_{kj}^{t+1} = \hat{\alpha}_{kj}^t - \lambda_t \sum_{i=1}^{n} \frac{\partial \ell(f(X_i), Y_i)}{\partial \alpha_{kj}}\bigg|_{\hat{\alpha}_{kj}^t}$$

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

Neural networks: Back-propagation

Returning to

$$\frac{\partial \ell(f(X_i), Y_i)}{\partial \beta_k} = -2(Y_i - f(X_i))Z_{ik}$$

$$\frac{\partial \ell(f(X_i), Y_i)}{\partial \alpha_{kj}} = -2(Y_i - f(X_i))\beta_k \sigma'(\alpha_{k0} + \alpha_k^\top X_i)X_{ij}$$

The gradients are updated via a two-pass algorithm:

(Known in general as back-propagation)

1. Forward pass: Current weights are fixed and all quantities are updated
2. Backward pass: The derivatives with respect to the weights are updated with the new updated quantities from the Forward pass.
3. These derivatives 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 the i^{th} observation, the algorithm can be run in either batch or online mode

Down sides:

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

  (We will return to dropout later)

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:

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

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 initial features 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)