STAT562 Lecture 17 Neural Network and Deep Learning

Beidi Qiang

SIUE

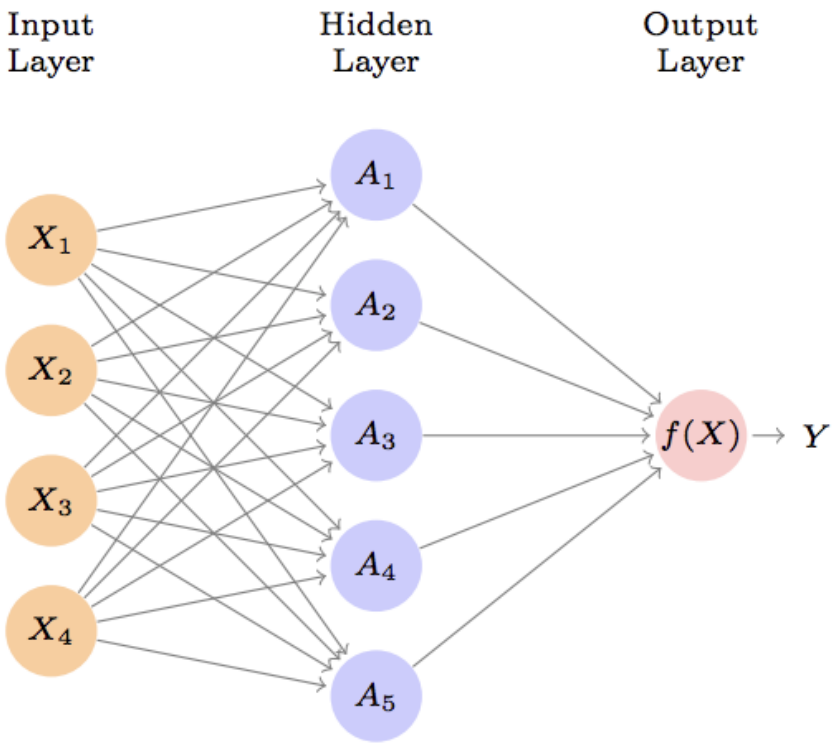
Introduction

Deep learning is a very active area of research in the machine learning and artificial intelligence. The cornerstone of deep learning is the neural network.

- Neural networks rose to fame in the late 1980s.
- Benefit from ever-larger training datasets, Neural networks start to gain popularity after 2010 with the new name deep learning and new architectures
- It is now considered one of the best model in problems such as image and video classification, speech and text modeling.

Single Layer Neural Networks

A neural network takes an input feature vector of $X = (X_1, X_2, ..., X_p)$ and builds a nonlinear function (structure) to predict the response Y. It has a unique structure:



Terminology of neural networks

- $X = (X_1, X_2, ..., X_p)$ forms the input layer.
- The arrows indicate that each of the inputs from the input layer feeds into each of the K hidden units.
- The K activations A_k in hidden layer are computed as functions (transformations) of the input features, i.e. $A_k = h_k(X)$.
- The output layer is a linear model (with quantitative labels) that uses these activations A_k as inputs, resulting in a function $f(X) = \beta_0 + \sum \beta_k A_k$.

Activation Function

At the hidden layer, A_k are computed as functions of X in the form

$$A_k = h_k(X) = g(\omega_{k0} + \omega_{k1}X_1 + \cdots + \omega_{kp}X_p)$$

- $g(\cdot)$ is a **nonlinear** "activation function" that is specified in advance.
- The nonlinearity in the activation function is essential, since without it the model would collapse into a simple linear model!
- All the parameters $\omega_{10}, \cdots, \omega_{Kp}$ in the activation function need to be estimated from training data.
- The $\beta_0 \cdots \beta_K$ in the output function also need to be estimated from training.

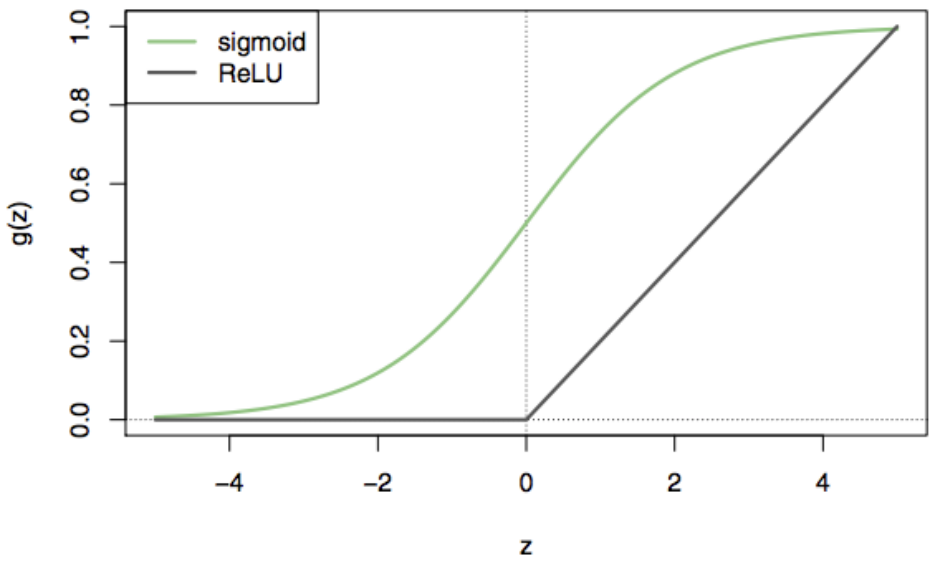
Common Choice of Activation Function

- Sigmoid:

$$g(z) = \frac{e^z}{1 + e^z}$$

- ReLU (rectified linear unit) [more preferred these days]:

$$g(z) = z * I(z > 0)$$



FYI...

The name neural network originally derived from thinking of these hidden units as analogous to neurons in the brain. Values of the activations A_k close to one are "firing", while those close to zero are "silent".

Estimating Coefficients

- Squared-error loss is used. The parameters are chosen to minimize

$$\frac{1}{2}\sum_{i=1}^{n}[y_i - f(x_i)]^2$$

where

$$f(x_i) = \beta_0 + \sum_{k=1}^{K}\beta_k g(\omega_{k0} + \sum_{j=1}^{p}\omega_{kj}x_{ij})$$

- The optimization problem is not simple given the nested parameters. Usually it is done using gradient descent and back-propagation. (Textbook section 10.7)

Weight Decay

Weight decay is a regularization technique by adding a small penalty, usually the L2 norm of the coefficients (weights), to the loss function.

- The parameters are chosen to minimize

$$\frac{1}{2}\sum_{i=1}^{n}[y_i - f(x_i)]^2 + \lambda\omega^T\omega$$

  where ω is the vector of all weights (coefficients) in the model.
- The regularization is added to prevent overfitting and to keep the weights small to avoid exploding gradient.

Multilayer Neural Networks

Modern neural networks typically have more than one hidden layer, and often many units per layer.

- $X = (X_1, X_2, \cdots, X_p)$ is the input layer.
- The activations $A_1^{(1)}, \cdots, A_{K_1}^{(1)}$ in first hidden layer is are computed base of the input layer, i.e.

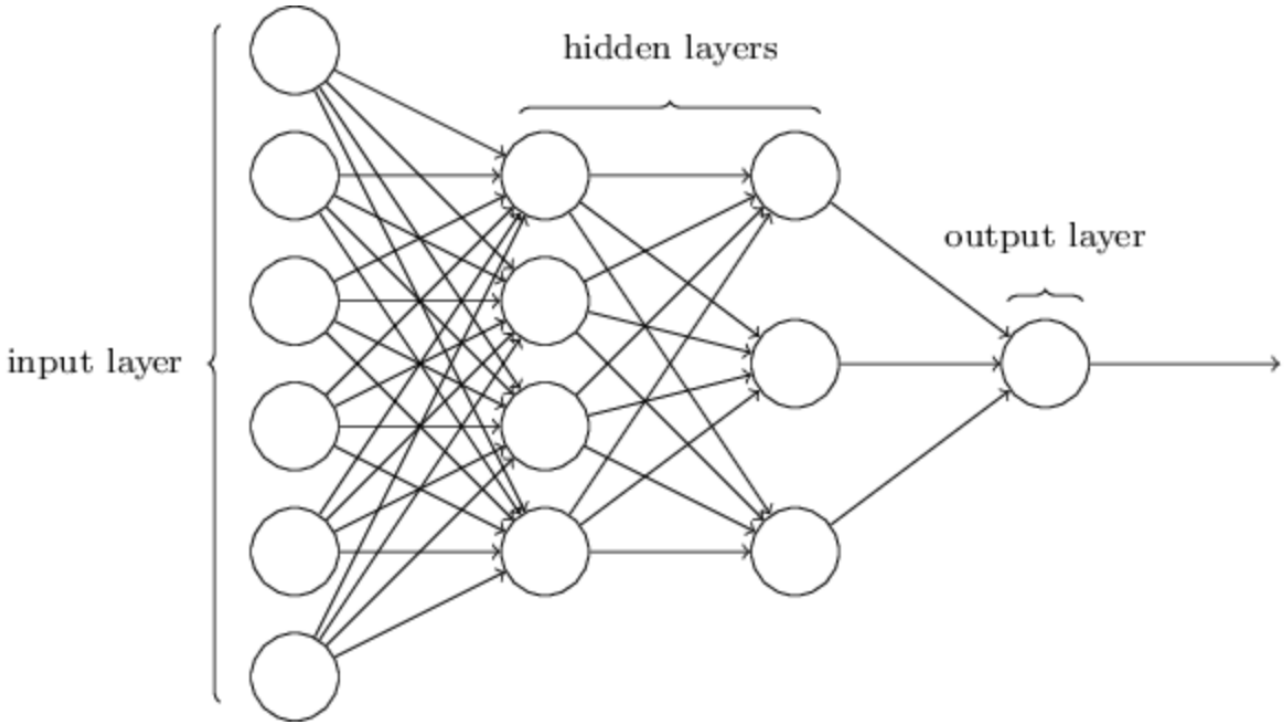
$$A_k^{(1)} = g(\omega_{k0}^{(1)} + \omega_{k1}^{(1)} X_1 + \cdots + \omega_{kp}^{(1)} X_p).$$

- The second hidden layer treats the activations $A_k^{(1)}$ of the first hidden layer as inputs and computes new activations

$$A_l^{(2)} = g(\omega_{l0}^{(2)} + \omega_{l1}^{(2)} A_1^{(1)} + \cdots + \omega_{lK_1}^{(2)} A_{K_1}^{(1)}).$$

- And so on...

2-Layer Neural Networks



Comments

- Each of the activations in the each of the hidden layers is a function of the input vector X.
- This is the case because they are function of previous layers and these go all way back to the input layer of X.
- Through a chain of transformations, the network is able to build up fairly complex transformations of X that ultimately feed into the output layer.

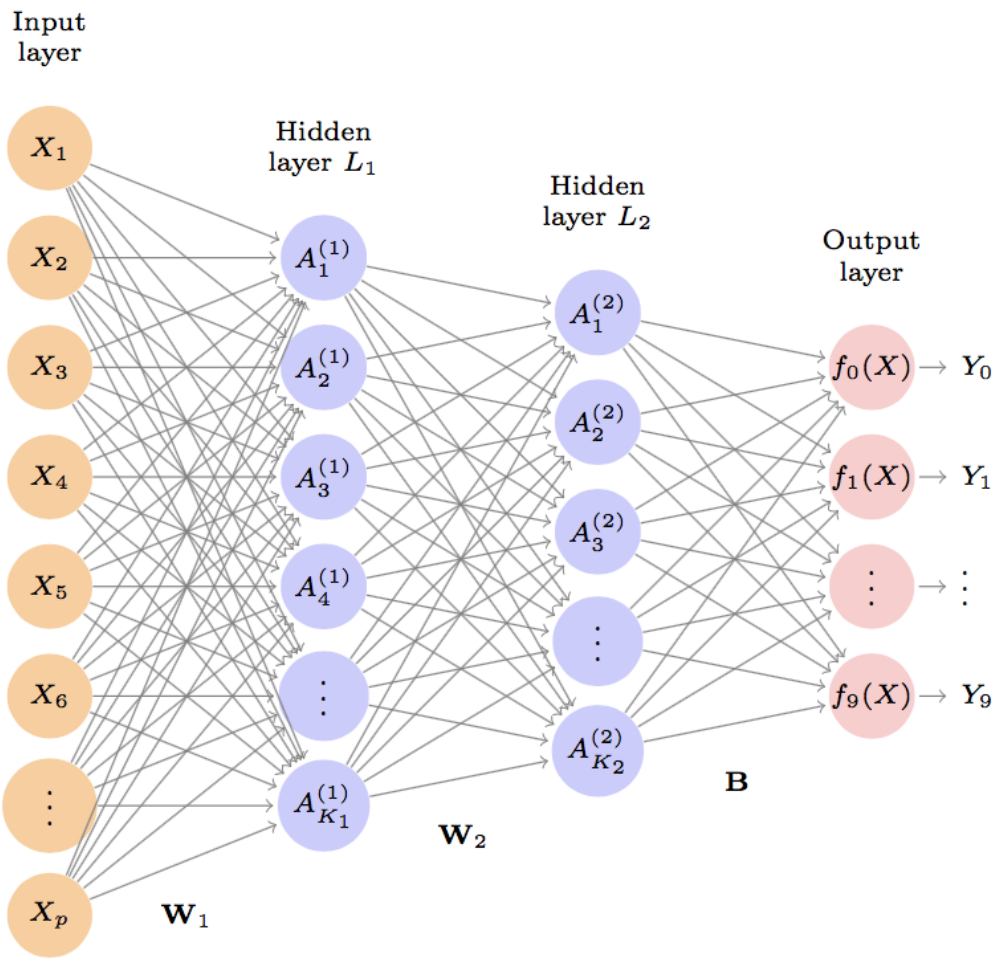
Activation function on the Output layer for Categorical Response

- When we have quantitative response, we simply compute $f(X) = \beta_0 + \sum \beta_l A_l^{(L)}$ at the output layer, where $A_l^{(L)}$ are the activations at the last hidden layer.
- In classification problems we would like our estimates to represent class probabilities. we use the special softmax activation function: Calculate $Z_m = \beta_0 + \sum \beta_l A_l^{(L)}$, for each class, $m = 0, 1, 2, \cdots, C$

$$f_m(X) = Pr(Y = m|X) = \frac{e^{Z_m}}{\sum_{i=0}^{C} e^{Z_i}}$$

- The classifier then assigns the class with the highest probability.

2-Layer Neural Networks in Classification



Example: Single Layer Nnet

Console | Terminal | Background Jobs

R 4.3.1 · ~/

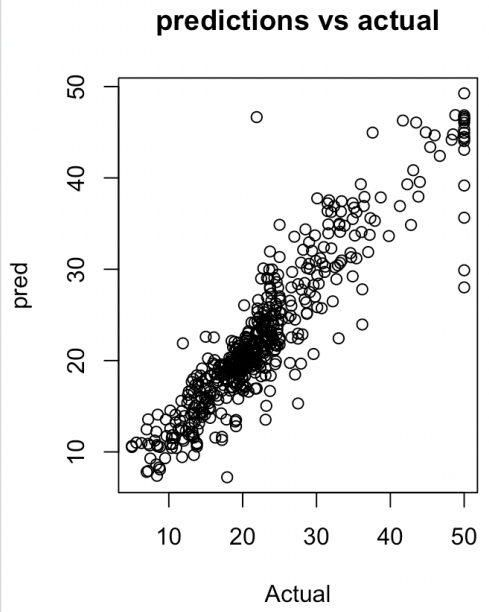
```
> library (nnet)
> nnet.fit=nnet(medv/50~ ., data=Boston, size=3,decay=0.1,linout=T)
# weights:  46
initial  value 17.069815
iter  10 value 15.559714
iter  20 value 12.277624
iter  30 value 9.202323
iter  40 value 8.112481
iter  50 value 5.864114
iter  60 value 3.451666
iter  70 value 3.125704
iter  80 value 3.026654
iter  90 value 2.974369
iter 100 value 2.933375
final  value 2.933375
stopped after 100 iterations
> pred=predict(nnet.fit)*50
> mse=mean((pred-Boston$medv)^2)
> plot(Boston$medv, pred, main="predictions vs actual", xlab="Actual")
> #Tunning
> library(caret)
> mygrid=expand.grid(.decay=c(0.1),.size=c(3,4,5,6,7,8))
> nnetfit=train(medv/50~., data=Boston, method="nnet", maxit=1000, tuneGrid=mygrid, trace=F)
> print(nnetfit)
Neural Network

506 samples
 13 predictor

No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 506, 506, 506, 506, 506, 506, ...
Resampling results across tuning parameters:

  size  RMSE        Rsquared   MAE
  3     0.08581603  0.7894123  0.06002583
  4     0.08445377  0.7947649  0.05842578
  5     0.08173036  0.8072009  0.05701671
  6     0.08189119  0.8071613  0.05685111
  7     0.08193556  0.8072438  0.05767213
  8     0.07942243  0.8189901  0.05604136
```

New Connection

Connection	Status

Files | Plots | Packages | Help | Viewer | Presen

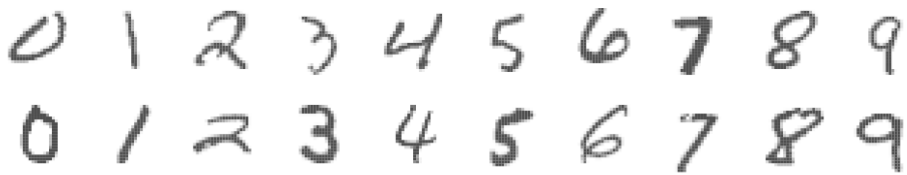
Zoom | Export

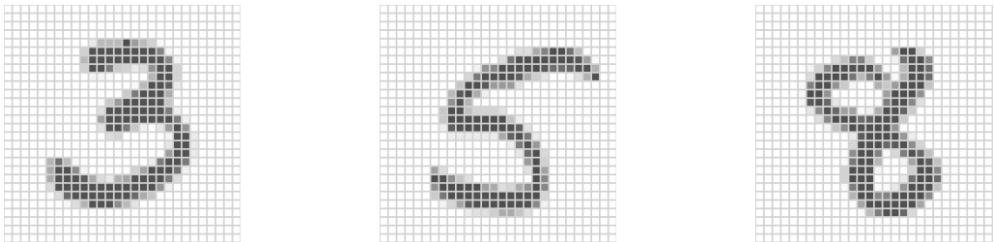


Example: MINST Data

We will illustrate a large feed-forward neural network on the famous MNIST handwritten digit dataset (available at http://yann.lecun.com/exdb/mnist).

- The goal is to build a model to classify the images into their correct digit class 0 to 9.
- Every image has $28X28 = 784$ pixels, each of which is an eight-bit grayscale value between 0 and 255 representing the relative amount of the written digit in that tiny square.





MINST Data Cont.

- There are 60,000 images in the training data and 10,000 in the test data.
- The first layer goes from 784 input units to a hidden layer of 256 units, which uses the ReLU activation function.
- The second hidden layer comes with 128 hidden units, also uses the ReLU activation function.
- The first hidden layer involves $(784 + 1) * 256 = 200960$ parameters and the second hidden layer involves $(256 + 1) * 128 = 32896$ parameters.
- The feed-forward ANN can be trained with Keras packages. The package is based on the TensorFlow backend engine and will require a python environment.
  see: https://www.python.org/downloads/
- Also worth noting are other Deep Learning packages in R, such as H2O, mxnet, deepnet and darch.

For implementation: see the Keras sample code here:
https://cran.r-project.org/web/packages/keras/vignettes/