Data Science

Deriving Knowledge from Data at Scale

Wee Hyong Tok

31 May 2017

Neural Networks

Artificial Neural Networks (ANN)

X_1	X_2	X_3	Y
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	0
0	1	0	0
0	1	1	1
0	0	0	0

Output Y is 1 if at least two of the three inputs are equal to 1.

Artificial Neural Networks (ANN)

X_1	X_2	X_3	Y
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	0
0	1	0	0
0	1	1	1
0	0	0	0

$$Y = I(0.3X_1 + 0.3X_2 + 0.3X_3 \quad 0.4 > 0)$$

$$\text{where } I(z) = \begin{matrix} 1 & \text{if } z \text{ is true} \\ 0 & \text{otherwise} \end{matrix}$$

Artificial Neural Networks (ANN)

- Model is an assembly of inter-connected nodes and weighted links;
- Output node sums up each of its input value according to the weights of its links;
- Compare output node against some threshold t

Perceptron Model

$$Y = I(\sum_i w_i X_i \quad t) \quad \text{or}$$

$$Y = sign(\sum_i w_i X_i \quad t)$$

Perceptron

- One of the earliest ML algorithms (Rosenblatt 1958).
- On-line linear binary classification algorithm.
- Determines a hyperplane (line in $\mathbb{R}^2$, plane in $\mathbb{R}^3$,…) separating the points for the two classes.

Linearly separable data:

Non-linearly separable data:

General Structure of ANN

Training ANN means learning the weights of the neurons

- Perceptrons only have two layers: the input layer and the output layer
- Perceptrons only have one output unit;
- Multi-layer ANNs consist of an input layer, hidden layer, and output layer
- Multi-layer neural networks can have several output units.

Multi-Layer Artificial Neural Networks

- The units of the hidden layer function as input units to the next layer
- However, multiple layers of linear units still produce only linear functions
- The step function in perceptrons is another choice, but it is not differentiable, and therefore not suitable for gradient descent search
- Solution: the sigmoid function, a non-linear, differentiable threshold function

How A Multi-Layer Neural Network Works

- The **inputs** to the network correspond to the attributes measured for each training tuple
- Inputs are fed simultaneously into the units making up the **input layer**
- They are then weighted and fed simultaneously to a **hidden layer**
- The number of hidden layers is arbitrary, although usually only one
- The weighted outputs of the last hidden layer are input to units making up the **output layer**, which emits the network's prediction
- The network is **feed-forward**: None of the weights cycles back to an input unit or to an output unit of a previous layer
- From a statistical point of view, networks perform **nonlinear regression**: Given enough hidden units and enough training samples, they can closely approximate any function

Feedforward Phase of Learning

Define the Network Topology

- Decide the network topology: Specify # of units in the ***input layer***, **# of *hidden layers*** (if > 1), # of units in ***each hidden layer***, and # of units in ***output layer***
- **Normalize** the input values for each attribute measured in the training tuples to [0.0 - 1.0]
- One **input** unit per domain value, each initialized to 0
- **Output**, if for classification and more than two classes, one output unit per class is used
- Once a network has been trained and its accuracy is **unacceptable**, repeat the training process with a ***different network topology*** or a ***different set of initial weights***

Math Fact

The gradient of the error: $\nabla E(\vec{\boldsymbol{w}}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_d}\right]$

(a vector in weight space) specifies the direction of the argument that leads to the steepest increase for the value of the error.

The negative of the gradient gives the direction of the <u>steepest decrease</u>.

Algorithm for Learning ANN

- Initialize the weights (w_0, w_1, …, w_k)

- Adjust the weights in such a way that the output of ANN is consistent with class labels of training examples
  - Objective function:

$$E = \sum_i \left[Y_i - f(w_i, X_i)\right]^2$$

  - Find the weights w_i's that minimize the above objective function
    - e.g., backpropagation algorithm

Backpropagation

- Iteratively process a set of training examples & compare the network's prediction with the actual known target value
- For each training example, the weights are modified to **minimize the mean squared error** between the network's prediction and the actual target value
- Modifications are made in the "**backwards**" direction: from the output layer, through each hidden layer down to the first hidden layer, hence "**backpropagation**"
- Steps
  - Initialize weights to small random numbers, associated with biases
  - Propagate the inputs forward (by applying activation function)
  - Backpropagate the error (by updating weights and biases)
  - Terminating condition (when error is very small, etc.)

Artificial Neural Networks

Known samples (historical data) are used to “train” the network.

Input data (x_i) are assigned weights (w_i) and combined in the “hidden” layer – like a set of linear regressions. These sets can then combined in additional layers – like regressions of regressions.

The sum of data and weights are transformed (“squashed”) to the range of the training data and error is measured.

A supervised training algorithm uses output error to adjust network weights to minimize errors.

Artificial Neural Networks

Stochastic Gradient Descent

- Converging to a minimum can be quite slow (i.e. it can take thousands of steps). Increasing the learning rate can improve this but can lead to overstepping minima
- If there are multiple local minima in the error surface, gradient descent can get stuck in one of them and not find the global minimum
- Gradient descent updates weights after summing over all training examples
- Stochastic gradient descent alleviates these difficulties
  - Randomly shuffle the data set;
  - Run small batch of training samples through the algorithm;
  - Stochastic (or incremental) gradient descent updates weights incrementally after calculating error for batch;
  - Continue until end of training set is reached
  - Related to Noise, Jitter, Simulated Annealing, etc.

Practical Considerations

- A good Backpropagation net requires more than the core of the learning algorithms. Many parameters must be carefully selected to ensure a good performance.
- Although the deficiencies of Backpropagation nets cannot be completely cured, some of them can be eased by some practical means.
- Initial weights (and biases)
  - Random, [-0.05, 0.05], [-0.1, 0.1], [-1, 1]
  - Normalize weights for hidden layer (v_ij) (Nguyen-Widrow)
    - Randomly assign v_ij for all hidden units V_j
    - For each V_j, normalize its weight by $\boldsymbol{v}_{ij} = \boldsymbol{v}_{ij} / \left\| \boldsymbol{v}_{\cdot j} \right\|_2$ and $= 0.7\sqrt[n]{\boldsymbol{p}}$

      where $\left\| \boldsymbol{v}_{\cdot j} \right\|_2$ is the normalization factor

      where $\boldsymbol{p} = \#$ of hiddent nodes, $\boldsymbol{n} = \#$ of input nodes
    - Avoid bias in weight initialization: $\left\| \boldsymbol{v}_{\cdot j} \right\|_2 =$ after normalization

Training samples

- Quality and quantity of training samples determines the quality of learning results
- Samples must be <u>good representatives</u> of the problem space
  - Random sampling
  - Proportional sampling (with prior knowledge of the problem space)
- # of training patterns needed:
  - There is no theoretically idea number. Following is a rule of thumb
  - W: total # of weights to be trained (depends on net structure)

  e: desired classification error rate
  - If we have P = W/e training patterns, and we can train a net to correctly classify (1 – e/2)P of them,
  - Then this net would (in a statistical sense) be able to correctly classify a fraction of 1 – e input patterns drawn from the same sample space
  - Example: W = 80, e = 0.1, P = 800. If we can successfully train the network to correctly classify (1 – 0.1/2)*800 = 760 of the samples, we would believe that the net will work correctly 90% of time with other input.

Over-Training/Over-Fitting

- Trained net fits very well with the training samples, but not with new input patterns
- Over-training may become serious if
  - Training samples were not obtained properly
  - Training samples have noise

Control over-training for better generalization

- Either divide the samples into two sets
  - - 90% into training set: used to train the network
  - - 10% into test set: used to validate training results

  periodically test the trained net with test samples, stop training when test results start to deteriorating, or
  - Use **Cross-validation**

- Stop training early
- Add noise to training samples: x:t becomes x+noise:t

Hyperparameters

Gradient descent

- Initial weights
- Learning rate schedule
- Batch size
- Momentum
- Stopping criteria

Tuning

- Split data set into training, cross-validation (cv), and test
- Fit model on training set
- Tune hyperparameters on Cross-Validation set
- Evaluate on test set

Some problems with backpropagation

The amount of information that each training case provides about the weights is at most the log of the number of possible output labels.

- So to train a big net we need lots of labeled data.

In nets with many layers of weights the backpropagated derivatives either grow or shrink multiplicatively at each layer.

- Learning is tricky either way.

Dumb gradient descent is not a good way to perform a global search for a good region of a very large, very non-linear space.

- So deep nets trained by backpropagation are **rare** in practice.

A solution to all of these problems

- Use greedy unsupervised learning to find a sensible set of weights one layer at a time. Then fine-tune with backpropagation.
- Greedily learning one layer at a time scales well to really deep networks.
- Most of the information in the final weights comes from modeling the distribution of input vectors.
  - The precious information in the labels is only used for the final fine-tuning.
- We do not start backpropagation until we already have sensible weights that already do well at the task.
  - So the fine-tuning is well-behaved and quite fast.

Neural Network as a Classifier

Strength

- High tolerance to noisy data
- Ability to classify untrained patterns
- Well-suited for continuous-valued inputs *and outputs*
- Successful on an array of real-world data, e.g., hand-written letters
- Algorithms are inherently parallel
- Techniques have been developed for the extraction of rules from trained neural networks

Weakness

- Long training time
- Require a number of parameters typically best determined empirically, e.g., the network topology or "structure."
- Poor interpretability: Difficult to interpret the symbolic meaning behind the learned weights and of "hidden units" in the network

Neural Networks - Example

Credits: Based on the example by Frauke Günther and Stefan Fritsch

Data

```
> head(infert)
  education age parity induced case spontaneous stratum pooled.stratum
1     0-5yrs  26      6       1    1           2       1             3
2     0-5yrs  42      1       1    1           0       2             1
3     0-5yrs  39      6       2    1           0       3             4
4     0-5yrs  34      4       2    1           0       4             2
5    6-11yrs  35      3       1    1           1       5            32
6    6-11yrs  36      4       2    1           1       6            36
```

Infert - Contains data of a case-control study that investigated infertility after spontaneous and induced abortion (Trichopoulos et al., 1976).

248 observations, 83 women, who were infertile (cases), and 165 women, who were not infertile (controls).

```r
library(neuralnet)

head(infert)

## Train a Neural network with two hidden neurons
nn <- neuralnet(
  case~age+parity+induced+spontaneous,
  data=infert, hidden=2, err.fct="ce",linear.output=FALSE)

##nn
##Visualize the neual network
plot(nn)


nn$result.matrix

covaresult <- cbind(nn$covariate,nn$net.result[[1]])
dimnames(covaresult) <- list(NULL,c("age","parity","induced","spontaneous","nn-output"))
head(covaresult)
```

Error: 140.667036 Steps: 3078

- Now change “hidden” parameter to 5
- What do you observe in the performance and results?

SVM vs. Neural Network

SVM

- Deterministic algorithm
- Nice generalization properties
- Hard to learn – learned in batch mode using quadratic programming techniques
- Using kernels one can learn very complex functions (hypothesis space)

Neural Network

- Nondeterministic algorithm
- Generalizes well but doesn't have strong mathematical foundation
- Can easily be learned in incremental fashion
- To learn complex functions—use multilayer perceptron (nontrivial)

Data Science

Deriving Knowledge from Data at Scale

That's all for tonight….