九阴真经

Yiping Ma

April 8, 2018

Contents

List of Figures

List of Tables

Preface

一住行窝几十年，蓬头长日走如颠。
海棠亭下重阳子，莲叶舟中太乙仙。
无物可离虚壳外，有人能悟未生前。
出门一笑无拘碍，云在西湖月在天。

–王重阳

Structure of book

You might want to add short description about each chapter in this book. Each unit will focus on <SOMETHING>.

About the companion website

The website[1] for this file contains:

- A link to latest version of this document.
- Link to download LaTeX source for this document.
- Miscellaneous material (e.g. suggested readings etc).

Acknowledgements

- A special word of thanks goes to ShareLatex[2](for LaTeX).

[1] https://Hazard1701@bitbucket.org/Hazard1701/long_term_notes.git
[2] https://www.sharelatex.com/templates/books/easy-book

1

Econometrics

说到算数，天下又有谁算得过我了？

– 《射雕英雄传》

1.1 Cross sectional regression

天之道，损有余而补不足。是故虚胜实，不足胜有余。

– 《射雕英雄传》

1.2 Time series regression

前途既已注定了是忧患伤心，不论怎生走法，终究避不了、躲不开，便如是咱们在长岭上遇雨一般。

– 《射雕英雄传》

1.3 Panel regression

手握灵珠常奋笔，心开天籁不吹箫。

– 《射雕英雄传》

2

Deep Learning

可怜未老头先白，春波碧草，晓寒深处，相对浴红衣

– 《射雕英雄传》

2.1 Model architecture

- How many hidden units should be used?
  - Suppose the inputs are M-dimensional, there are D units in the hidden layer.
  - If D > M, it's mapping inputs to a higher dimensional space, just the usage of kernel in SVM.
  - If D < M, it's a kind of (nonlinear) dimension reduction.
  - D = 1 and it's the final output, it's simply a logistic regression.
- How many layers should be used?
  - Theoretically, a one hidden layer neural net can approximate any continuous functions.
  - Practically, deep networks are easier to train than shallow networks. Typically practitioners build a deep model first (maybe deep enough to capture any complex relationships, as deep as the computer can afford), then use regularization to avoid overfitting.

- Activation functions
  - **Sigmoid** The most classical, but not widely used in modern ANN. Its drawbacks include:
    - Vanishing gradients (neurons saturated).
    - Output not centered at 0.
  - **Tanh** Like sigmoid, with range extended to [-1, 1]. Tanh is almost always better than sigmoid as the output is centered at 0. One exception is for the output layer, when working with classification (as the range of sigmoid is a natural for probabilities).
  - **ReLu** Very popular in deep learning. If you don't know what to use, start with ReLu (default one). ReLu helps with vanishing gradient problems, but some of its gradients may die and never be activated again. There is also soft version of ReLu, e.g., log(exp(x)+1).
  - **Leaky ReLu** Fix the "dead neuron" problem of ReLu.
  - Never use linear activation in hidden units. You may use linear activation for the output layer for regression purpose (the range of output is $\mathbb{R}$).
  - There is no clear rule for choosing activation function. A good activation function should have domain (where the function is sensitive) consistent with the range of its input tensor, while producing tensor whose range is consistent with the domain of the next layer's activation function.
- Loss functions
  - Classification: cross-entropy
  - Regression: mean squared error
- Other hyperparameters
  - The choice of hyperparameters could depend on the application domains (e.g., NLP, computer vision, ...), the configuration of

your computer, the sample size, and so on. In general neural network modelling is a very iterative process where one has to go over idea-code-experiment again and again. It's almost impossible to get the best hyperparameters at the very first attempt.

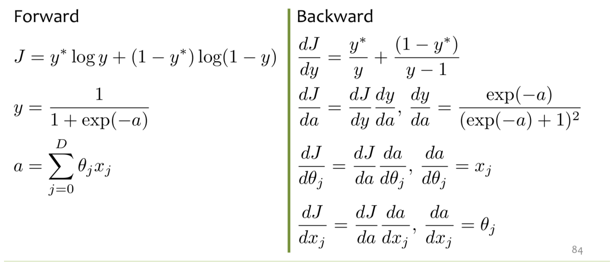
- **The number of layers, the number of nodes in each layer**, choice of **activation function** are discussed in 2.1.
- **The number of epochs** Increase the number of epochs until the validation accuracy starts decreasing even when training accuracy is increasing(overfitting). The number of epochs usually doesn't matter that much when early stopping scheme is introduces.
- **Batch size** A good default for batch size might be 32. Also try 32, 64, 128, 256, and so on.
- **Learning rate** Large learning rate speeds up the learning process but may result in oscillating after certain steps. Using decaying learning rate is typically preferred.

2.2 Backpropagation

- The algorithm
  - Forward computation
    1. Write an algorithm for evaluating the function y = f(x). The algorithm defines a directed acyclic graph, where each variable is a node (i.e. the "computation graph")
    2. Visit each node in topological order. For variable u_i with inputs $v_1, ..., v_N$,
       a. Compute $u_i = g_i(v_1, ..., v_N)$
       b. Store the result at the node
  - Backward computation
    1. Initialize all partial derivatives dy/du_j to 0 and $dy/dy = 1$.
    2. Visit each node in reverse topological order.
    For variable $u_i = g_i(v_1, ..., v_N)$
       a. We already know dy/du_i

b. Increment dy/dv_j by $(dy/du_i)(du_i/dv_j$ (for each u_i, we shall visit each v_j which are connected with u_i in the next layer)

- An example of backpropagation for logistic regression



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-a)}$$

$$a = \sum_{j=0}^{D} \theta_j x_j$$

Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

$$\frac{dJ}{da} = \frac{dJ}{dy}\frac{dy}{da}, \quad \frac{dy}{da} = \frac{\exp(-a)}{(\exp(-a) + 1)^2}$$

$$\frac{dJ}{d\theta_j} = \frac{dJ}{da}\frac{da}{d\theta_j}, \quad \frac{da}{d\theta_j} = x_j$$

$$\frac{dJ}{dx_j} = \frac{dJ}{da}\frac{da}{dx_j}, \quad \frac{da}{dx_j} = \theta_j$$

84

Figure 2.1: Backpropagation for logistic regression

- Why Backpropagation is fast
  - There are only two types of functions in neural network models, i.e., linear and activation functions.
  - Derivatives of linear functions are straight forward. We pick activation functions y=f(x) whose derivatives can be expressed as simple functions in y.
  - During the forward pass, we have computed all relevant variables which would be used in the backward pass.

2.3 Training tips

- **Weights initialization** If you initialize all weights to zeros, all hidden units will be identical during the training, which makes it equivalent to ”one node” at all layers. This problem is known as ”failing to break symmetry”. Random initialization fixes this. For random initialization, we usually keep initial parameters small so that activation functions are sensitive at this region. Otherwise you are exposed to vanishing gradient problems. A good example of initialization approach is ”He initialization”, where weights are random standard normal variables scaled by 2/(dimension of the previous layer).

- **Cut train/validation/test datasets** For relatively small datasets, a 60/20/20 split of train/validation/test sets is usually desirable. However, when you have millions of observations and a sample of 10,000 points is enough for fair performance evaluation purposes, a 98/1/1 split could be better than 60/20/20.
- **Dev/test sets should be consistent** Sometimes in order to enlarge sample size, you may explore training data whose distribution is not completely the same as that of test data. This is fine as long as you make sure that validation data and test data have the same distribution.
- **Learning curve**
  - A large discrepancy between validation and training set errors indicates high variance (overfitting). Possible solutions include enlarging datasets or adding regularizations.
  - A large training set error (relative to the base error rate) may indicate high bias (underfitting). Possible solutions include adding complexities to the model architecture.
  - If the learning curve is oscillating, it could be that the learning rate is too big/gradients are too large. Clipping gradient norm, reducing learning rate, or a better learning rate decay scheme may help.
- **Mini batch gradient descent**
  - **Batch gradient descent** Too slow per each iteration.
  - **SGD** Lose the speed up advantage of vectorization.
  - **Mini batch gradient descent**
    - Mix of batch gradient descent and SGD.
    - Better performance for most of the time, especially for large datasets.
    - Typical batch sizes include 64, 128, 256, 512. This is also a hyperparameter which can be optimized. (It's better to be the power of 2 due to the way computer memory is layed out)

- **Gradients update schemes**
  - Momentum gradient descent
    - Formula

$$\begin{aligned}&V_{dw} = \beta V_{dw} + (1-\beta)dW, \quad V_{db} = \beta V_{db} + (1-\beta)db\\&w := w - \alpha V_{dw}, \quad b := b - \alpha V_{db}\end{aligned}$$

    - Motivation: When the cost function contour is not spherically symmetric (in high dimensional space), the original gradient descent path may oscillate on the ”minor” axis. Momentum helps partially fix this problem by speeding up the movement on the major axis while slowing down the movement on the minor axis. Momentum gradient descent can be thought of as an exponential moving average of the original gradient, which reduces oscillation.
  - RMSprop
    - Formula

$$\begin{aligned}&S_{dw} = \beta S_{dw} + (1-\beta)\underbrace{dW^2}_{elementwise}, \quad S_{db} = \beta S_{db} + (1-\beta)\underbrace{db^2}_{elementwise}\\&w := w - \alpha\frac{dw}{\sqrt{S_{dw}+\epsilon}}, \quad b := b - \alpha\frac{db}{\sqrt{S_{db}+\epsilon}}\end{aligned}$$

    - Motivation: w is the major axis (where there is a long way from starting point to the minimum) and b is the minor axis (where there is no much difference between starting point and minimum). However, when the cost function contour is a ellipse, the gradient’s projection on b is large while the projection on w is small. Thus by dividing $\sqrt{S_{dw}+\epsilon}$ or $\sqrt{S_{db}+\epsilon}$ we amplify the gradient on w direction. The ϵ here is just for numerical purposes. (avoid zero denominator)
  - Adam optimization

* Formula

$$V_{dw} = \beta_1 V_{dw} + (1-\beta_1)dW, \quad V_{db} = \beta_1 V_{db} + (1-\beta_1)db \xleftarrow{\text{momentum}}$$

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2)dW^2, \quad S_{db} = \beta_2 S_{db} + (1-\beta_2)db^2 \xleftarrow{\text{RMSprop}}$$

$$V_{dw}^{corrected} = \frac{V_{dw}}{1-\beta_1^t}, \quad V_{db}^{corrected} = \frac{V_{db}}{1-\beta_1^t} \xleftarrow{\text{bias correction}}$$

$$S_{dw}^{corrected} = \frac{S_{dw}}{1-\beta_2^t}, \quad S_{db}^{corrected} = \frac{S_{db}}{1-\beta_2^t} \xleftarrow{\text{bias correction}}$$

$$w := w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \epsilon}$$

$$b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \epsilon}$$

* Motivation: This is a combination of momentum and RMSprop algorithms, which is proven to be effective on many deep learning applications. All of momentum, RMSprop, Adam can help learning algorithms get rid of local minimums and saddle points.
* Choice of hyperparameters:
  · α: needs to be tune.
  · β_1: start with 0.9
  · β_2: start with 0.999
  · ϵ: 10^{-8}

- **Learning rate decay methods**
  - $\alpha = \frac{1}{1+\text{decay rate}*\text{epoch num}}\alpha_0$
  - $\alpha = 0.95^{\text{epoch num}}\alpha_0$
  - $\alpha = \frac{k}{epochnum}\alpha_0$
  - Stepwise functions. (Restricting the number of times that *alpha* can decay is also a way to realize early stopping)

- **Normalizing features** Make the cost function contour more symmetric so that the gradient descent can proceed faster.

- **Vanishing/exploding gradients** The gradients can explode/vanish exponentially fast with respect to the number of layers. Proper initialization of weights helps with this problem. For instance, for a single neuron with ReLu activation, we can make w_i be drawn from normal distribution with variance 1/n where n is the number of inputs for this neuron. Clipping gradient norm may also help.

2.4 How to avoid over-fitting

- L2 regularization (much more widely used than L1 regularization in neural networks)

$$J(w^1, b1, ..., w^L, b^L) = \frac{1}{m}\sum_{i=1}^{m} L(y^i, \hat{y^i}) + \frac{\lambda}{2m}\sum_{l=1}^{L} ||w^l||_F^2$$

  where

$$\begin{aligned} ||w^l||_F^2 &= \sum_{i=1}^{n[l]}\sum_{j=1}^{n[l-1]} (w_{ij}^{[l]})^2 \\ dw^{[l]} &= (\text{from backprop}) + \frac{\lambda}{m} w^{[l]} \\ w^{[l]} &= w^{[l]} - \alpha dw^{[l]} \\ &= \underbrace{(1 - \alpha\frac{\lambda}{m})w^{[l]}}_{\text{weight decay}} - \alpha(\text{from backprop}) \end{aligned}$$

- **Dropout** Randomly drop a certain fraction of nodes at each layer. Multiply the weights of the remaining layer by (1/fraction remained). It helps prevent over-fitting because the dropout process prevents the situation that one node is being relied on too much. When weights are more spread out on different nodes, the Frobenius norm of the weight matrix decreases. Dropout is not applied on the test set.
- **Data Augmentation** Flip, rotate, zooming in, etc.
- **Early stopping**
  - **Advantage** You only have to go through gradient descent process once without trying lots of different values for λ.

  - **Disadvantage** The task of minimizing cost function and the task of regularization are not orthogonal. (The algorithm learns better when these two tasks are well separated)

- Reduce the architecture complexity to make it more generalizable.

2.5 Convolution neural networks

- **Step by Step**
  Input matrix $\xrightarrow{convolution}$ feature maps $\xrightarrow{activation function}$activated feature maps
  $\xrightarrow{pooling}$pooling layers$\xrightarrow{\dots}$...

- **filter/kernel/feature detector** You will slide the filter over the original input matrix to compute a matrix called ”convolved feature/activation map/feature map”.

- **convolved feature/activation map/feature map** See 2.2.

- **Attributes of feature map**

  - *Depth* The number of filters we used for convolution operation. If depth=n, the feature map will be n stacked matrices.
  - *Stride* The number of ”cell” (e.g., pixels) by which we slide the filter over the input matrix.
  - *Zero-padding* Whether to pad the input matrix with zeros around the boarder. Zero padding allows us to control the size of the feature map.

- **Pooling/Sub-sampling**:For example, in case of max pooling, we define a spatial neighborhood and take the largest element from the activated(e.g., rectified) feature map within this neighborhood. We can also do average pooling, etc. Pooling is a method to reduce dimensionality while retaining the most important information.

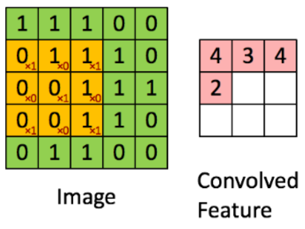


Figure 2.2: The convolution operation. The green matrix is the original input matrix. The yellow matrix is the filter. The pink output is the convolved feature.

2.6 Interpretation

- **Neural networks and brain** ”A single neuron in the brain is an incredibly complex machine that even today we don’t understand. A single “neuron”in a neural network is an incredibly simple mathematical function that captures a minuscule fraction of the complexity of a biological neuron. So to say neural networks mimic the brain, that is true at the level of loose inspiration, but really artificial neural networks are nothing like what the biological brain does.” –Andrew Ng

- **Why CNN** Convolution operation is a way to extract low-level features from the input. When the stride is larger than 1, it has downsampling effect. (The activation map now has lower dimensions than the input) It’s sort of like PCA, where we use a low dimensional representation to capture the most important characteristics of the original input matrix. Convolution is also a way to capture local dependencies.