Deep Learning with R

Chenshu Liu

April 2022

Contents

DL 01 Regression as a first step in deep learning

Prediction from linear regression

```
# Scenario: predicting sales performance
sales <- c(3, 4, 2, 4, 5, 6, 3, 9, 1, 12)
```

In the simplest way possible, we can simple find the mean of all the observations, and use the mean as our prediction for future sales performances

$$Performance = \frac{\Sigma_{i=1}^{n} x_i}{n}$$

```
mean.sales <- mean(sales)
mean.sales
```

```
## [1] 4.9
```

How to measure error in prediction

However, using only the mean as future prediction is not close to accurate, we are bound to make errors. Thus, to measure the magnitude of the error we are making using the mean, we introduce the concept of *sum of squared errors (SSE)*:

$$SSE = \Sigma_{i=1}^{n}(x_i - \bar{X})^2$$

Where $\bar{X}$ is the mean of the observations.

NOTE: There the differences are squared because there can be positive and negative numbers in the differences, and summing positive and negative differences together will make 0. Thus, we need to **square** the differences to make them significant.

```
SSE = sum((sales - mean.sales)^2)
SSE
```

```
## [1] 100.9
```

However, the measurement of deviation from actual observations has some limitations:

1. In the calculation of SSE, we summed all the differences together, which means the sample size plays a role in the calculation of the deviation. Thus, in order to rule out the effect of sample size on SSE, we introduce the idea of **variance**: $var = \frac{\Sigma_{i=1}^{n}(x_i - \bar{X})^2}{n-1}$, where $n-1$ is the degrees of freedom. Now, the standard deviation is a square-root version of the measurement of deviation, which makes more sense.

2. Because the differences are squared, it is hard for us to interpret the deviation, thus, we further introduce the idea of *standard deviation*: $s = \sqrt{\frac{\Sigma_{i=1}^{n}(x_i - \bar{X})^2}{n-1}}$

Both the variance and standard deviation describes how bad the model's prediction is

```r
variance <- (sum((sales - mean.sales)^2))/(length(sales) - 1)
variance
```

```
## [1] 11.21111
```

```r
# base R function
var(sales)
```

```
## [1] 11.21111
```

```r
standard.deviation <- sqrt((sum((sales - mean.sales)^2))/(length(sales) - 1))
standard.deviation
```

```
## [1] 3.3483
```

```r
# base R function
sd(sales)
```

```
## [1] 3.3483
```

So, predictions are bound to have errors:

$$y_i = \bar{X} + \epsilon_i$$
$$target = model + error$$

Take away: After learning about how to measure the accuracy of a prediction model, we now need to know how to optimize the model and letting it to generate more accurate predictions. The task of optimizing the model that minimizes prediction error is the in the scope of Deep Learning!

How to potentially reduce prediction error

Assume that we have a base-line linear regression model that has a SSE we call *sst* (unsystematic variance), if we tune the slope and intercept of the base-line linear regression model, it has a new SSE we call *ssm* (systematic variance). Thus, in order to **measure the improvement in prediction accuracy brought by tuning the parameters in the model**, we introduce the measurement of R-squared:

$$R^2 = \frac{ssm}{sst}$$

R^2 describes the amount of variance that can be described by the new, improved model, with respect to the base-line model.

Preview on next section

Thought question: we know that tuning the parameters in the model can lead to improvement in the prediction performance, but how can we find the optimal parameter that can minimize the prediction error (**core** goal in deep learning). This is what we will be covering in the next section

Reference video: https://www.youtube.com/watch?v=0F2bBZiirlg&list=PLH5_eZVldmtUCZWp-eL0lVL7SA6qyDIf9&index=1

DL 02 Linear regression as a Simple Learner "SL"

Related functions/calculations in prediction

Prediction function

From last section, we introduced that predictions can be made with a linear function in the form of:

$$\hat{y_i}(x_i) = \beta_0 + \beta_1 x_i$$

Where $\hat{y}$ is the predicted value based on the prediction function

Loss function

Loss function is used to measure the amount of error in **one of** our predictions using the model:

$$L(x_i) = [\hat{y_i}(x_i) - y_i]^2$$

The loss function is just the SSE that we introduced in the last section, where we take the square of the differences between observed and predicted value, at the ith position (because there are many observations and predictions)

Cost function

The cost function is a little from the loss function because the cost function is calculated from the persepective of the overall prediction, instead of each individual prediction's deviation (calculated by loss function):

$$\begin{aligned} C(\beta_0, \beta_1) &= \frac{1}{n}\Sigma_{i=1}^{n} L \\ &= \frac{1}{n}\Sigma_{i=1}^{n}[\hat{y_i}(x_i) - y_i]^2 \\ &= \frac{1}{n}\Sigma_{i=1}^{n}[\beta_0 + \beta_1 x_i - y_i]^2 \end{aligned}$$

How to find the optimial parameters

Say we find a cost function that is:

$$\begin{aligned} C &= \frac{1}{5} \times [\beta_0 + \beta_1(1.3) - 0.7]^2 + \cdots + [\beta_0 + \beta_1(3.3) - 3.5]^2 \\ &= 6.55 - 4.68\beta_0 + \beta_0^2 - 13.132\beta_1 + 5.08\beta_0\beta_1 + 7.002\beta_1^2 \end{aligned}$$

We can see that the cost function is composed of the two parameters β_0 and β_1, and we also know that we want to minimize the cost function. So, our goal now is to find optimal values of β_0 and β_1 so that the cost function is at its minimum. The way to achieve so, its through using **partial derivatives**

$$\frac{\partial C}{\partial \beta_0} = 2\beta_0 + 5.08\beta_1 - 4.68 \tag{1}$$

$$\frac{\partial C}{\partial \beta_1} = 5.08\beta_0 + 14.004\beta_1 - 13.132 \tag{2}$$

By solving (1) and (2), we can obtain the following augmented matrix for the linear system:

$$\begin{bmatrix} 2 & 5.08 & 4.68 \\ 5.08 & 14.004 & 13.132 \end{bmatrix}$$

Then, we can reduce the augmented matrix which give us the final values for $\beta_0 = -0.532267$ and $\beta_1 = 1.13081$

Gradient descent

The case above is a two dimensional (having to variables), which is rather simple. However, in real-life scenarios, we often have many parameters and we still need to find the set of parameters that minimizes the cost function, so we ought to find a more generalized way to find optimal parameters, so we introduce the idea of **gradient descent**

1. Randomly choose a point in space and find the derivative (i.e. slope) of the cost function at that particular point: $slope_x$
2. Define a learning rate (LR), which is a predefined value for gradient descent
3. Calculate step: $x_{new} = x - LR \times slope_x$
4. Repeat the same process from step1-3 for x_{new}

Reference video: https://www.youtube.com/watch?v=FrceOv_oJac&list=PLH5_eZVldmtUCZWp-eL0lVL7SA6qyDIf9&index=2

DL 03 Linear regression as a Shallow Neural Network "SNN"

Multiple linear regression in R

```r
df <- read.csv("MultipleLinearRegression.csv")
df
```

```
##      x1   x2  x3     y
## 1  20.1 39.3 1.3 394.5
## 2  23.6 31.6 1.5 211.4
## 3  29.2 36.9 1.4 251.4
## 4  29.3 34.1 1.2  85.4
## 5  30.0 37.2 1.2 248.6
## 6  22.9 39.3 1.9  46.0
## 7  25.1 33.0 1.3 252.5
## 8  27.7 36.0 2.0 315.4
## 9  24.7 34.5 1.3 120.5
## 10 24.2 39.8 1.5 110.1
```

After preliminary view of the data, we can see that there are three independent variables x_1, x_2, x_3, so we are dealing with a multiple linear regression:

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 \approx y$$

Now, there are four parameters (i.e. $\beta_0, \beta_1, \beta_2, \beta_3$) that we need to tune, in order to optimize prediction, so we have the loss function for the four parameters:

$$L^{(i)}(\beta_0, \beta_1, \beta_2, \beta_3) = (\beta_0 + \beta_1 x_1^{(i)} + \beta_2 x_2^{(i)} + \beta_3 x_3^{(i)} - y^{(i)})^2$$

```r
# multiple linear regression
mlr <- lm(y ~., data = df)
summary(mlr)
```

```
##
## Call:
## lm(formula = y ~ ., data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -151.55  -96.65   22.22   56.09  164.10
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 342.7300   777.7247   0.441    0.675
## x1           -3.5850    14.2899  -0.251    0.810
## x2            0.2326    16.6230   0.014    0.989
## x3          -38.0145   166.3937  -0.228    0.827
##
## Residual standard error: 134.4 on 6 degrees of freedom
## Multiple R-squared:  0.01687,    Adjusted R-squared:  -0.4747
## F-statistic: 0.03433 on 3 and 6 DF,  p-value: 0.9906
```

Single layer neural network

Based on the multiple linear regression we conducted in last subsection, we are actually getting into neural network layers. The multiple linear regression itself can be considered as a single layer network (Figure 1: Single Layer Neural Network), where the inputs are taken into hidden layers and multiplied with the weights (the parameters), and then output the prediction in the output layer

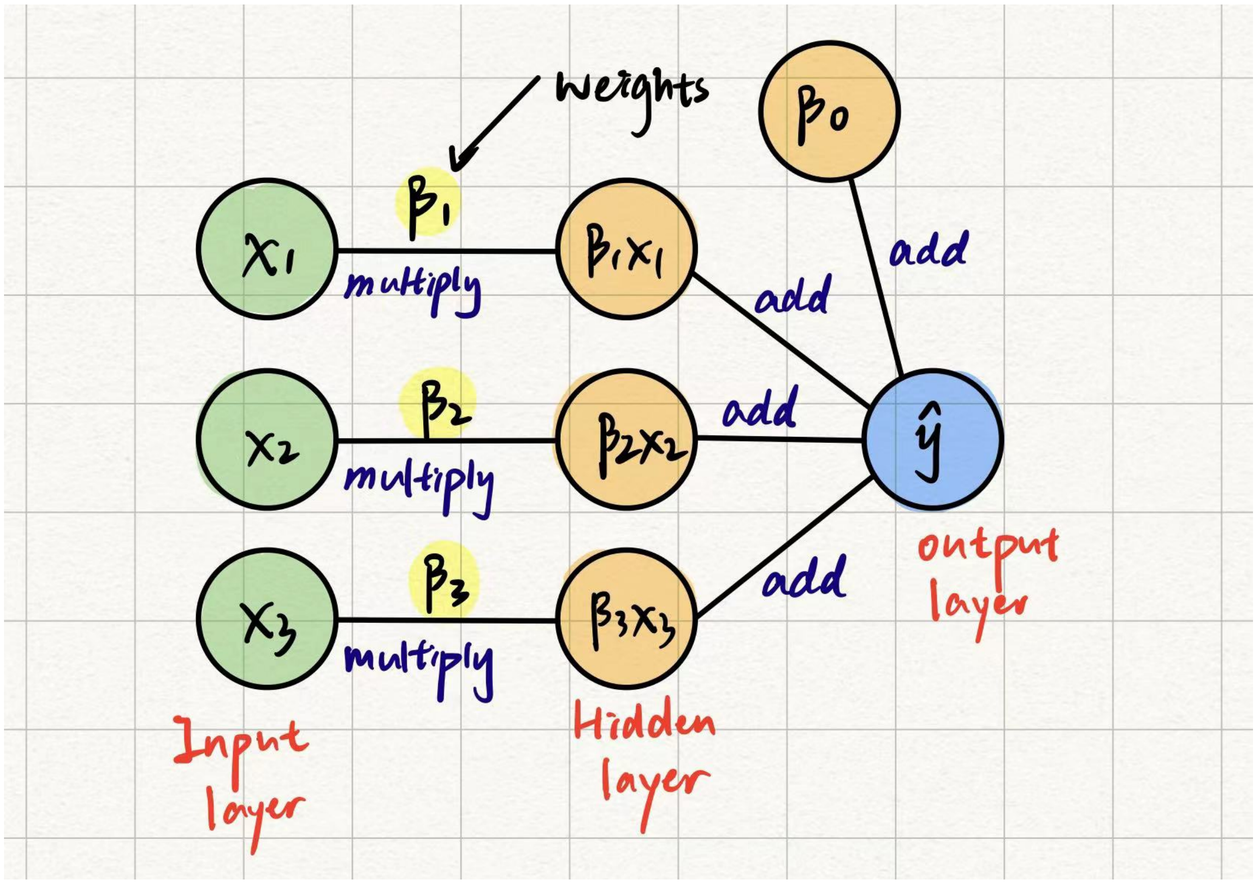

Figure 1: Single Layer Neural Network

Now, here's when things get interesting:

1. We can use **forward propagation** to find output prediction (i.e. feeding in input values, and calculate for output)
2. We can also use **backward propagation** to better tune the parameters (i.e. using the predicted values, we can go back to tune the parameters and allow for better prediction performance)

Reference video: https://www.youtube.com/watch?v=ZX4YSidnQaI&list=PLH5_eZVldmtUCZWp-eL0lVL7SA6qyDIf9&index=6

DL 04 Logistic regression as a Neural Network