Binary logistic regression modeling with TensorFlow™

Zhongheng Zhang[1], Lei Mo[2], Chen Huang[3], Ping Xu[4]; written on behalf of AME Big-Data Clinical Trial Collaborative Group

[1]Department of Emergency Medicine, Sir Run Run Shaw Hospital, Zhejiang University School of Medicine, Hangzhou 310016, China; [2]Department of Biostatistics, Lejiu Healthcare Technology Co., Ltd, Shanghai, China; [3]Nursing Department, Information Technology (IT) Center, Sir Run Run Shaw Hospital, Zhejiang University School of Medicine, Hangzhou 310016, China; [4]Emergency Department, Zigong Fourth People's Hospital, Zigong 643000, China

Correspondence to: Zhongheng Zhang. No. 3, East Qingchun Road, Hangzhou 310016, China. Email: zh_zhang1984@zju.edu.cn.

Abstract: Logistic regression model is one of the most widely used modeling techniques in clinical medicine, owing to the widely available statistical packages for its implementation, and the ease of interpretation. However, logistic model training requires strict assumptions (such as additive and linearity) to be met and these assumptions may not hold true in real world. Thus, clinical investigators need to master some advanced model training methods that can predict more accurately. TensorFlow™ is a popular tool in training machine learning models such as supervised, unsupervised and reinforcement learning methods. Thus, it is important to learn TensorFlow™ in the era of big data. Since most clinical investigators are familiar with the logistic regression model, this article provides a step-by-step tutorial on how to train a logistic regression model in TensorFlow™, with the primary purpose to illustrate how the TensorFlow™ works. We first need to construct a graph with tensors and operations, then the graph is run in a session. Finally, we display the graph and summary statistics in the TensorBoard, which shows the changes of the accuracy and loss value across the training iterations.

Keywords: Logistic regression; TensorFlow; gradient descent

Submitted Sep 17, 2019. Accepted for publication Sep 18, 2019.
doi: 10.21037/atm.2019.09.125
View this article at: http://dx.doi.org/10.21037/atm.2019.09.125

Introduction

Binary logistic regression modeling is probably one of the most commonly used approaches for predictive analytics in clinical medicine. The advantage of this modeling technique is that its estimated coefficient is easy to understand. The exponentiation of the coefficient gives the odds ratio, which is directly interpretable for clinicians (1). Furthermore, there are many statistical packages available for the implementation of the logistic regression modeling. The limitation of the logistic regression approach is that it cannot automatically model complex relationships among covariates such as non-linear and interaction terms. In the era of big data, numerous feature variables are readily available from electronic healthcare records, and it is usually challenging for researchers to correctly specify the model with domain knowledge. Many sophisticated machine learning algorithms have been developed to deal with such high-dimension data. The advantage of these advanced algorithm is that they can model complex relationship among feature variables without explicitly specifying interactions and high-order terms (2,3). The limitation is that they are black-box approaches that the causal relationship between variables and labels are not easily understandable for subject matter audience (4).

The training of prediction models heavily relies on TensorFlow™ in modern era in the business domain. TensorFlow™ is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The advantages of TensorFlow™ include: (I) good computational graph

visualization; (II) efficient library management backed by Google; and (III) execution subparts of a graph allows to retrieve discrete data onto an edge and therefore offers great debugging method. However, TensorFlow™ has not been widely used in clinical research partly due to the technical complexity of its implementation. Due to many advantages of the TensorFlow™, the present article aims to introduce TensorFlow™ by illustrating how to train a logistic regression model.

Working example

We first create a dataset for the illustration purpose. The data are generated by a function called *sim_data()*. The dataset includes five feature variables, namely, *age*, *lac*, *wbc*, *sex* and *type*. The *mort* was the outcome (label). The training set is *dat* and the testing set is the *dat_test*.

```
> sim_data <- function(n=2000){
library(dummies)
age <- round(abs(rnorm(n,mean = 60, sd = 20)))
lac <- round(abs(rnorm(n,3,1)),1)
wbc <- round(abs(rnorm(n,10,3)),1)
sex <- factor(rbinom(n,size = 1,prob = 0.6),
labels = c("Female","Male"));
type <- as.factor(sample(c("Med","Emerg","Surg"),
size = n,replace = T,
prob = c(0.4,0.4,0.2)))
linPred <- cbind(1,age,lac,wbc,dummy(sex)[,-1],
dummy(type)[,-1]) %*%
c(-30,0.2,4,1,-2,3,-3)
pi <- 1/(1+exp(-linPred))
mort <- factor(rbinom(n,size = 1, prob = pi),
labels = c("Alive","Died"))
dat <- data.frame(age=age,lac=lac,wbc=wbc,
sex=sex,type=type,
mort = mort)
return(dat)
}
> set.seed(123)
> dat <- sim_data()
> dat_test <- sim_data(n=1000)
```

After running the above code, we can take a look at the data frame:

```
> head(dat)
```

	age	lac	wbc	sex	type	mort
1	49	2.5	10.6	Male	Emerg	Alive
2	55	3.2	12.0	Male	Emerg	Died
3	91	2.5	12.0	Female	Med	Died
4	61	4.2	6.1	Female	Emerg	Died
5	63	3.2	3.9	Male	Surg	Alive
6	94	2.4	16.6	Female	Surg	Died

There are three numeric variables including *age*, *lac* and *wbc*; and two categorical variables that are *sex* and *type* (also called factor variable in R). There are two levels for the outcome variable *mort*: Alive and Died.

Training logistic regression model with conventional method

Logistic regression model can be trained by using the build-in R function glm(), which is used to fit generalized linear models, specified by giving a symbolic description of the linear predictor and a description of the error distribution.

```
> mod <- glm(mort~., data = dat, family = "binomial")
> library(tableone)
> ShowRegTable(mod,exp = F)
```

	coef	[confint]	p
(Intercept)	-28.92	[-32.46, -25.73]	<0.001
age	0.19	[0.17, 0.22]	<0.001
lac	3.94	[3.49, 4.44]	<0.001
wbc	0.94	[0.82, 1.07]	<0.001
sexMale	-1.96	[-2.41, -1.52]	<0.001
typeMed	2.70	[2.19, 3.24]	<0.001
typeSurg	-2.81	[-3.40, -2.25]	<0.001

The above output shows the coefficients estimated by using maximum likelihood method. All coefficients are statistically significant with P values less than 0.001. Next, we will show how the model performs in the test dataset. Note that the test dataset is not used for training the model.

```
> pred <- predict.glm(mod,newdata = dat_test,
type = "response")
> library(pROC)
> roc(response = dat_test$mort,predictor = pred,ci=T)
```

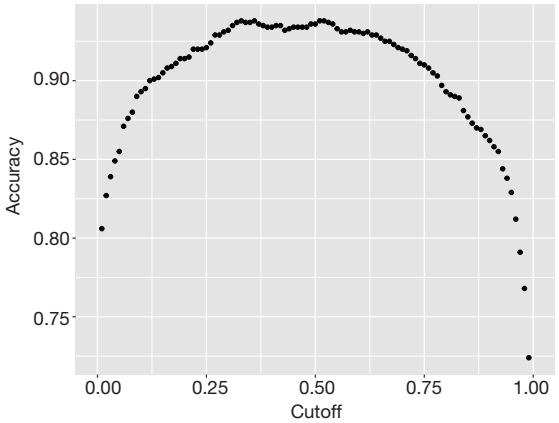

Figure 1 Scatter plot showing the changes of the model accuracy with the cutoff values for determining survivors versus non-survivors. It appears that the accuracy reaches its maximum at the cutoff value of 0.5.

```
Call:
roc.default(response = dat_test$mort, predictor = pred, ci = T)

Data: pred in 314 controls (dat_test$mort Alive) < 686 cases
(dat_test$mort Died).
Area under the curve: 0.9841
95% CI: 0.9784-0.9897 (DeLong)
> predBi <- pred >= 0.5
> crossTab <- table(predBi,dat_test$mort)
> (crossTab[1]+crossTab[4])/sum(crossTab)
[1] 0.936
```

The area under the characteristic curve (AUROC) is a standard measure to assess the discrimination of a model. In our example, the AUROC is 0.9841, indicating that the model performs perfectly in discriminating survivors and non-survivors. The accuracy (0.936) is another measure for the performance of the model, which is obtained by dividing the correctly classified subjects by the total number of subjects. Note that the accuracy is dependent on the cutoff values used to judge *Alive* versus *Died* subjects. Thus, we can plot the accuracy against the cutoff values to examine the relationship between the two quantities.

```
> DTaccuracy <- data.frame()
> for (cutoff in seq(0,1,by = 0.01)) {
predBi <- pred >= cutoff;
crossTab <- table(predBi, dat_test$mort)
accuracy = (crossTab[1]+crossTab[4])/sum(crossTab)
DTaccuracy <- rbind(DTaccuracy,c(accuracy,cutoff))
}
> names(DTaccuracy) <- c('Accuracy','Cutoff')
> qplot(x=Cutoff, y = Accuracy, data = DTaccuracy)
```

We vary the cutoff value by a step of 0.01 and calculate the accuracy at each cutoff value. The output *Figure 1* shows that the accuracy is the highest at the cutoff value of 0.5, which means that subjects who predicted to have a probability of death greater than 0.5 by the training model should be judged as *Died*; otherwise, they are predicted to be *Alive*.

TensorFlow™ method

Splitting the data into training and validation cohort

In machine learning practice, the dataset is usually split into the training and validation sets. The purpose of the validation set is to tune hyperparameters such as the learning rate, number of batches and epochs. More advanced algorithms such as neural networks can have more hyperparameters including the number of hidden layers and weight decay (5). However, the latter ones are out of the scope of the present discussion. Because the validation set is used to tune hyperparameters, it contributes to the model training process (i.e., the model sees the validation data during training). Thus, we also need a testing dataset to verify that the trained model is generalizable to future data.

```
> library(caret)
> y = with(dat, model.matrix(~ mort + 0))
> x = with(dat,
model.matrix(~ age + lac+wbc+sex+type))[,-1]
> trainIndex = createDataPartition(1:nrow(x),
p=0.7, list=FALSE,times=1)

> x_train = x[trainIndex,]
> x_valid = x[-trainIndex,]
> y_train = y[trainIndex,]
> y_valid = y[-trainIndex,]
```

In the example, we use the model.matrix() function to generate a design (or model) matrix, by expanding

factors to a set of dummy variables (depending on the contrasts) and expanding interactions similarly. Then the createDataPartition() is employed to split the dataset into the training and validation samples by the ratio of 7:3.

Data dimension

It is very important to clarify the data dimension within the TensorFlow™ framework. In our example, each subject is a 1×6 vector for the feature space, and the outcome (label) is a 1×2 vector. The number of classes is 2.

```
> dim(x_train)
[1] 1400  6
> dim(y_train)
[1] 1400  2
```

Placeholders for features and labels

Placeholder is one of the tensor types used in TensorFlow™. It is a variable that we will assign data in a future time. Placeholders are nodes whose value is fed in at execution time. In the example, placeholders refer to the features and labels that will be used in a session, they are empty in the graph.

```
> library(tensorflow)
> tf$reset_default_graph()
> X <- tf$placeholder(tf$float32,
shape(NULL, ncol(x)),
name = "X")
> Y = tf$placeholder(tf$float32, shape(NULL, 2L),
name = "Y")
```

The above code firstly loads the *tensorflow* package (version: 1.13.1.9000) to the workspace. Details for the installation of TensorFlow within R environment are available at https://tensorflow.rstudio.com/tensorflow/articles/installation.html. The tf$reset_default_graph() function clears the default graph stack and resets the global default graph. The tf$placeholder() function has three arguments: data type, shape and name. In the example, the data type is float32 for both features and labels. The shape is the dimension of the features and labels. The NULL value in the shape means that the first dimension of the placeholder can be any number of subjects. The *name* argument specifies the Tensorflow name that will appear in the graph.

TensorFlow™ variables for weights and bias

TensorFlow™ variables are stateful nodes which output their current value; meaning that they can retain their value over multiple executions of a graph. It is the best way to represent shared, persistent state manipulated by your program. In fact, variables are the things that you want to tune in order to minimize the loss, such as the bias and weights in the example.

```
>W = tf$Variable(tf$random_normal(shape(ncol(x),2L),
stddev = 1.0),
name = "weights")
>b = tf$Variable(tf$zeros(shape(2L)), name = "bias")
```

TensorFlow™ variables can be created using the tf$Variable() function. The arguments define the shape and initial values of the variables. The property of the variable W can be viewed with the following code:

```
> print(W)
<tf.Variable 'weights:0' shape=(6, 2) dtype=float32_ref>
```

Unlike the conventional S3 R object, the result of a TensorFlow™ object cannot be obtained until running a session.

Operations for logistic regression model

Binary logistic regression model requires a sigmoid function to transform the probability into the logit scale.

$$\mathrm{P}(\mathrm{mort}=1)=\frac{1}{1+e^{-z}}$$

where $z = b + w_1 \cdot age + w_2 \cdot lac + w_3 \cdot wbc + w_4 \cdot sexMale + w_5 \cdot typeMed + w_6 \cdot typeSurg$

Instead of using the conventional mean squared error, we use a cost function called Cross-Entropy, also known as Log Loss. Cross entropy consists of two parts: one for mort =1 and the other for mort = 0. A cost function basically tells us how good our model is at making predictions for a given value of W and b.

$$\text{loss} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \cdot log\left(\hat{y}_i\right) + \left(1-y_i\right) \cdot log\left(1-\hat{y}_i\right)\right]$$

where y_i is the observed outcome which takes values of 0 or 1; $\hat{y}_i$ is the predicted probability of event taking values from 0 to 1 ($i.e\ \hat{y}_i = \frac{1}{1+e^{-z_i}}$). With cross entropy, as the predicted probability comes closer to 0 ($\hat{y}_i \mapsto 0$) for the "yes" example $(y_i = 1)$, the loss increases closer to infinity. The purpose of model training is to find appropriate weights (W) and bias (b) to minimize the loss function.

```
> logits = tf$add(tf$matmul(X, W), b)
> pred = tf$nn$sigmoid(logits)
```

The tf$matmul() function multiply two matrix X and W, note that the second dimension of X must be equal to the first dimension of W according to the matrix multiplication rule. Then the tf$add() function add the bias term b, resulting in a tensor in logit scale. The tfnnsigmoid() function is employed to transform the logit scale to probability. The next step is to define the loss function. We will use sigmoid cross entropy with logits as a loss function.

```
> entropy = tf$nn$sigmoid_cross_entropy_with_logits(labels = Y,
logits = logits)
> loss = tf$reduce_mean(entropy, name = "loss")
```

With the loss function being defined, we can use gradient descent approach to find appropriate weights and bias to minimize the loss function. The gradient measures how much the output of a function changes if you change the input a little bit. It can be thought of as the slope of a function. A higher gradient means a steeper slope and the faster a model can learn. In mathematical terms, a gradient is a partial derivative of the loss function with respect to its weights (6). The gradient of the loss function can be written as $\nabla_w L(w)$ w.r.t. the weights. The learning rate η determines the size of the step we take to reach the minimum. The update process can be written as: $w = w - \eta \cdot \nabla_w L(w)$. The process can be written in R code as follows:

```
> learning_rate = 0.01
> optimizer = tf$train$
GradientDescentOptimizer(learning_rate = learning_rate)$
minimize(loss)
> init_op = tf$global_variables_initializer()
```

The second line defines an operation to initialize global variables in the graph. Now that we have trained the model, let's evaluate it:

```
> correct_prediction <- tf$equal(tf$argmax(logits, 1L),
tf$argmax(Y, 1L),
name = "correct_pred")
> accuracy <- tf$reduce_mean(tf$cast(correct_prediction,
tf$float32),
name = "accuracy")
```

Save all values of model performance

There is a special operation called *summary* in TensorFlow™ to facilitate visualization of the model parameters like weights and biases of a logistic regression model, metrics like loss or accuracy values, and images like input images to a neural network. The summary operation takes in a regular tensor and outputs the summarized data to the computer disk.

```
> loss_scalar <- tf$summary$scalar("Loss", loss)
> accuracy_scalar <- tf$summary$scalar("Accuracy",
accuracy)
> W_hist <- tf$summary$histogram("Coefficient", W)
> b_hist <- tf$summary$histogram("Intercept", b)
> merged <- tf$summary$merge_all()
```

The tf$summary$scalar() function is to write the values of a scalar tensor that changes over time or iterations to the computer disc. In the example, the loss and accuracy of the model performance are scalar tensors that change over training iterations. Similarly, the tf$summary$histogram() function is used to plot the histogram of the values of a non-scalar tensor. This gives us a view of how does the histogram (and the distribution) of the tensor values change over training iterations. In the example, it's used to monitor the changes of weights and biases distributions. The last line of code merges all statistics so that all summaries can be run at once within the running session.

Execute a graph within a session

Now that we have structured the graph, let's execute a graph within a session. Note that all results can only be returned after running a session. Technically, a session places the graph operations on hardware such as CPUs or GPUs and

provides methods to execute them.

```
> learning_rate = 0.01
> with(tf$Session() %as% sess, {
sess$run(init_op)
for (i in 1:10000) {
sess$run(optimizer,
feed_dict = dict(X=x_train,Y=y_train))
}
sess$run(accuracy,
feed_dict=dict(X = x_valid,Y = y_valid))
})
[1] 0.758
```

The output shows that the accuracy is only 0.758, which is far less than that obtained by the conventional glm() function. We need to tune the hyperparameters to obtain a better predictive performance.

Hyperparameters

There are many hyperparameters in model training. Here we will explain some important ones for the logistic regression model.

- Epoch is one forward pass and one backward pass of all the training examples.
- Batch size is the number of training examples in one forward/backward pass. The larger the batch size, the more memory space you'll need.
- Iteration is one forward pass and one backward pass of one batch of subjects in the training set.

Now let's define some values for these important hyperparameters.

```
> epochs = 10000 # Total number of training epochs
> batch_size = 30 # Training batch size
> display_freq_iter = 10 # Frequency of displaying the training
results
> display_freq_epoch = 1000
> learning_rate = 0.01 # The optimization initial learning rate
```

The *display_freq_iter* and *display_freq_epoch* are parameters for printing results during the training iterations. They will not influence the results. The above specification results in a total of 46 iterations.

```
> nrow(x_train)/batch_size
[1] 46.66667
```

Create an interactive session

An alternative method to run a session is by calling the tf$InteractiveSession() function. The only difference with a regular Session is that an *InteractiveSession* installs itself as the default session on construction. In other words, the *InteractiveSession* supports less typing, as allows to run variables without needing to constantly refer to the session object. In the example, we launch an InteractiveSession:

```
> sess = tf$InteractiveSession()
```

TensorBoard

TensorBoard is a visualization tool that comes with any standard TensorFlow™ installation. In Google's words: "*The computations you'll use TensorFlow for (like training a massive deep neural network) can be complex and confusing. To make it easier to understand, debug, and optimize TensorFlow programs, we've included a suite of visualization tools called TensorBoard.*" The two most important purposes of TensorBoard is: (I) to visualize the graph and (II) writing summaries to visualize the learning process. For example, the changes of accuracy across training epochs can be visualized with TensorBoard.

To visualize the graph with TensorBoard, we need to write log files of the program. To write event files, we first need to create a *train_writer* for those logs, using this code:

```
> train_writer <- tf$summary$FileWriter(logdir = "YOUR
PATH/logs")
> train_writer$add_graph(sess$graph)
```

The directory stores log files and the graph of the program is added by using the add_graph() function. We will not call the tensorboard() function at present because we want to store more summary statistics to the train_writer object.

Run the model within a session

We need to initialize all variables at the beginning of running a session.


```
> sess$run(init_op)
# Number of training iterations in each epoch
> num_tr_iter = floor(nrow(y_train) / batch_size)
```

All used global variables need to be initialized (i.e., you do not need to initialize variables that are not run, or none of the runs depends on them.). The number of iterations is explained as above.

```
> for (epoch in 1:epochs){
if(epoch %% display_freq_epoch == 0){
print(paste('Training epoch:',epoch),quote = F)
}
# Randomly shuffle the training data at the beginning of each
#epoch
sample_seq <- sample(1:nrow(x_train))
x_train <- x_train[sample_seq,]
y_train <- y_train[sample_seq,]
for (iteration in 1:num_tr_iter) {
start = (iteration-1) * batch_size+1
end = iteration * batch_size
x_batch = x_train[start:end,]
y_batch = y_train[start:end,]
# Run optimization op (backprop)
feed_dict_batch = dict(X = x_batch,Y = y_batch)
result <- sess$run(list(merged, optimizer),
feed_dict=feed_dict_batch)
summary <- result[[1]]# extract the summary result of merged
train_writer$add_summary(summary, epoch) # write summary
#to disk
if(iteration %% display_freq_iter == 0 &
epoch %% display_freq_epoch == 0){
# Calculate and display the batch loss and accuracy
loss_batch = sess$run(loss,
feed_dict=feed_dict_batch)
acc_batch = sess$run(accuracy,
feed_dict=feed_dict_batch)
print(sprintf("iter %i: Loss=%.2f, Training Accuracy=%.2f",
iteration, loss_batch, acc_batch),quote = F)
}
}
# Run validation after every 100 epoch
if(epoch %% display_freq_epoch == 0){
feed_dict_valid = dict(X = x_valid,Y = y_valid)
loss_valid = sess$run(loss, feed_dict=feed_dict_valid)
acc_valid = sess$run(accuracy, feed_dict=feed_dict_valid)
print('---------------------------------------------------------',quote = F)
print(sprintf("Epoch: %i,
validation loss: %.2f, validation accuracy: %.2f",
epoch, loss_valid, acc_valid),quote = F)
print('---------------------------------------------------------',quote = F)
}
}
```

The above code loops through epochs. Recall that we have defined the total number of epochs to be 10,000. There are a number of iterations within each epoch. The actual data are passed to the sess$run() function by using the dict() function. The tensor objects (merged and optimizer) in the list argument are the part of the graph that run in the session. It is not necessarily to run the whole graph. The above loop print validation loss and accuracy at a frequency of 1,000 epochs, and at an iteration of 10 within each epoch. The output for the last training epoch is as follows:

```
[1] Training epoch: 10000
[1] iter 10: Loss=0.18, Training Accuracy=0.90
[1] iter 20: Loss=0.28, Training Accuracy=0.97
[1] iter 30: Loss=0.24, Training Accuracy=0.87
[1] iter 40: Loss=0.18, Training Accuracy=0.93
[1] ---------------------------------------------------------
[1] Epoch: 10000, validation loss: 0.18, validation accuracy: 0.92
[1] ---------------------------------------------------------
```

Model validation in the testing set

Note that the model is evaluated in the validation cohort in the above session run, we can validate the model performance in the testing set as follows:

```
> x_test = with(dat_test,
model.matrix(~ age + lac + wbc+
sex + type))[,-1];
> y_test = with(dat_test, model.matrix(~ mort + 0))
> feed_dict_test = dict(X = x_test, Y = y_test)
> loss_test = sess$run(loss, feed_dict = feed_dict_test);
```


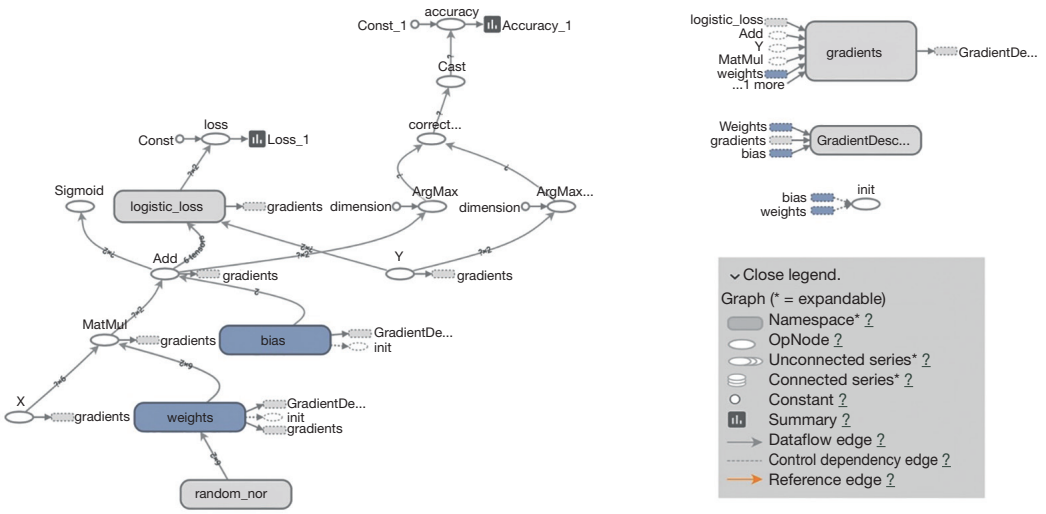

Figure 2 TensorBoard showing the graph of tensor flow for the logistic regression model.

```
> acc_test = sess$run(accuracy, feed_dict = feed_dict_test)
> print(sprintf("Test loss: %.2f, test accuracy: %.2f",
loss_test, acc_test),quote = F)
[1] Test loss: 0.19, test accuracy: 0.93
```

The first two lines transform the testing set to those suitable for passing to the TensorFlow™ placeholder. In the following sessions, we only run the loss and accuracy tensors because we already have trained the model with updated weights and bias. The result showed that the accuracy of the model was 0.93 in the testing set.

Weights and bias

The weights and bias after training can be obtained by running the *b* and *W* tensors in the session. Also remember to close the session after running.

```
> sess$run(b)
[1] 21.10329 -21.11448
> sess$run(W)
```

	[,1]	[,2]
[1,]	-0.1486220	0.1487045
[2,]	-2.9049671	2.9064085
[3,]	-0.6881254	0.6884889
[4,]	1.6609378	-1.6614350
[5,]	-1.9071095	1.9081335
[6,]	2.1673198	-2.1680720

```
> sess$close()
```

The output shows that the weights and bias are close to the one obtained by the glm() method. However, you need to tune the hyperparameters such as learning rate and the number of epochs to achieve the minimal loss.

Launch the TensorBoard

The tensorboard() function provides a tool to inspect and understand your TensorFlow runs and graphs. The argument *log_dir* is to specify the directory to scan for training logs.

```
> tensorboard(log_dir = "/YOUR PATH/logs")
```

The TensorFlow™ graph is shown in *Figure 2*. The graph displays the tensors and operations defined in the previous code. From the bottom, there is a matrix multiplication between X and weights (*W*), and then the bias (*b*) was added. The tensor shape is shown in the data flow edge. In the scalar tab of the TensorBoard, there are two plots showing the accuracy and loss across training epochs. It appears that the training accuracy and loss stabilize after 5,000 epochs (*Figures 3* and *4*).

Concluding remarks

The article shows how to perform logistic regression model

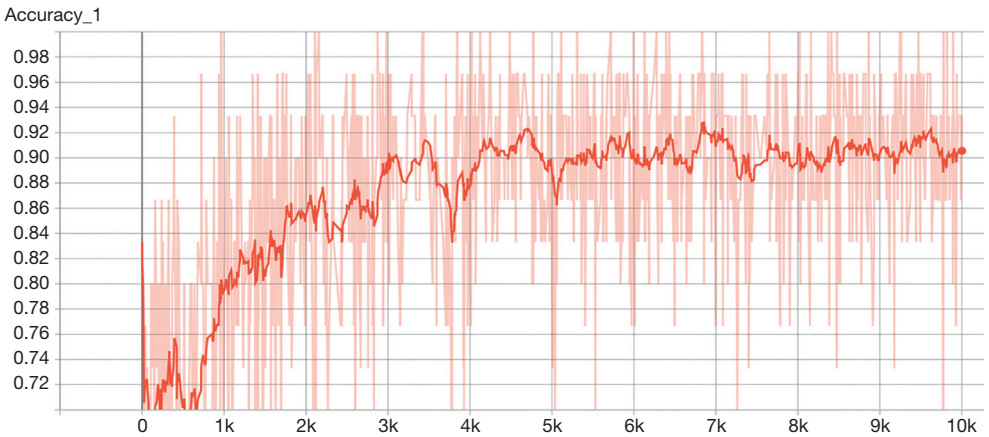

Figure 3 Training accuracy across epochs.

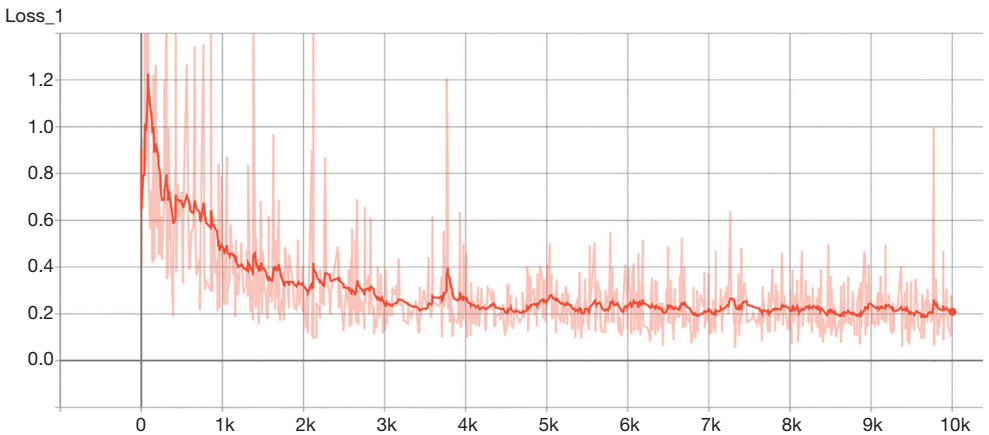

Figure 4 Training loss across epochs.

training in TensorFlow™ within R. It is useful for beginners who are going to work with TensorFlow™. However, the power of TensorFlow™ is not fully demonstrated with the current example (i.e., the accuracy of the model trained with TensorFlow™ is no better than the one trained with the conventional method). It is probably due to the fact that the data is simulated with generalized linear model and there is no complex interaction and non-linear terms. Thus, the maximum likelihood method is capable to obtain the weights and bias to maximize the likelihood function. Since the function is linear with monotone property, there is no local minima. TensorFlow™ has the power to model complex relationship among features and labels and it has been widely used for some deep learning methods. Understanding how the TensorFlow™ works will help to explore more on data science.

Acknowledgments

None.

Footnote

Conflicts of Interest: The authors have no conflicts of interest to declare.

Ethical Statement: The authors are accountable for all aspects of the work in ensuring that questions related to the accuracy or integrity of any part of the work are appropriately investigated and resolved.

References

1. Tolles J, Meurer WJ. Logistic Regression: Relating Patient

Characteristics to Outcomes. JAMA 2016;316:533-4.
2. Shi L, Wang XC. Artificial neural networks: Current applications in modern medicine. IEEE, 2010:383-7.
3. Chen JH, Asch SM. Machine Learning and Prediction in Medicine - Beyond the Peak of Inflated Expectations. N Engl J Med 2017;376:2507-9.
4. Castelvecchi D. Can we open the black box of AI? Nature 2016;538:20-3.
5. Stefaniak B, Cholewiński W, Tarkowska A. Algorithms of Artificial Neural Networks - Practical application in medical science. Polski Merkuriusz Lekarski. 2005;19:819-22.
6. Ruder S. An overview of gradient descent optimization algorithms. Vol. cs.LG, arXiv.org. 2016.

Cite this article as: Zhang Z, Mo L, Huang C, Xu P; written on behalf of AME Big-Data Clinical Trial Collaborative Group. Binary logistic regression modeling with TensorFlow™. Ann Transl Med 2019;7(20):591. doi: 10.21037/atm.2019.09.125

