Chapter 4
Initialization of Network Parameters

A thought is a great big vector of neural activity and they have causal powers.

Geoffrey Hinton

Abstract In this section, we will learn how initialization of the parameters affects a neural network model. We will explore different initialization techniques and visualize the results.

We will be using the following R packages:

```
library(ggplot2)
library(gridExtra)
library(InspectChangepoint)
```

4.1 Initialization

Weight initialization can have a profound impact on both the **convergence** rate and the **accuracy** of our network. While working with deep neural networks, initializing the network with the correct weights can make the difference between the network converging in a reasonable time and the loss function "oscillating", and not going anywhere even after thousands of iterations.

To understand why is initialization a problem, let us consider the sigmoid function represented in Fig. 4.1. The `sigmoid` activation function is approximately linear when we are close to zero (represented by the red line in the figure). This means that as the weights tend toward zero, **there will not be any nonlinearity**, which goes against the very ethos and advantage of deep layer neural networks.

© Springer Nature Singapore Pte Ltd. 2019
A. Ghatak, *Deep Learning with R*,
https://doi.org/10.1007/978-981-13-5850-0_4

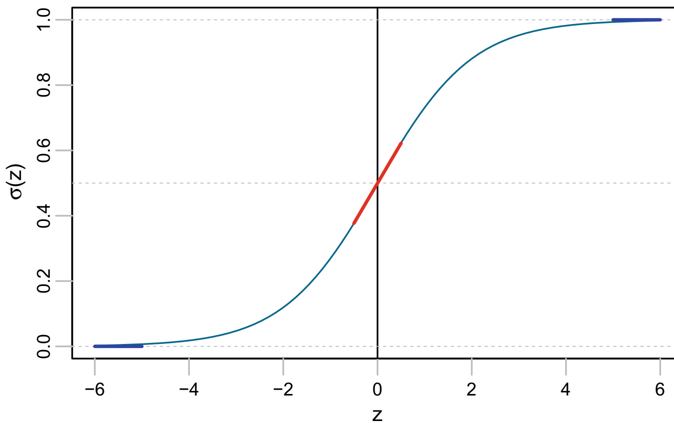


Fig. 4.1 The sigmoid function becomes "flat", at the extremes and "linear" in the middle

If the weights are too small, then the variance of the input signal starts diminishing as it passes through each layer in the network and eventually drops to a very low value, which is no longer useful.

If the weights are too large, then the variance of input data tends to increase with each passing layer and at some point of time, it becomes very large. For very large values, the `sigmoid` function tends to become flat (represented by the blue line in the figure) as we can see in Fig. 4.1. This means that our activations will become saturated and the gradients will start approaching zero.

Therefore, initializing the network with the right weights is very important if we want our neural network to function properly and make sure that the weights are in a reasonable range before we start training the network.

In order to illustrate this fact, let us use our DNN application with four different weight initializations

(a) Initialize weights to zero.
(b) Initialize weights to a random normal distribution $\mathcal{N}(0, \mu = 0, \sigma = 1)$.
(c) The weights are initialized to a random normal distribution but inversely proportional to the square root of the number of neurons in the previous layer (`Xavier` initialization).
(d) Initialize weights to a random normal distribution but inversely proportional to the square root of the number of neurons in the previous layer and directly proportional to the square root of 2 (`He` initialization).

A well-designed initialization can **speedup the convergence** of gradient descent and increase the chances of a lower training and generalization error. Let us go through four different types of parameter initializations and what difference do they make on the cost and convergence.

Before we do that, let us create a spirally distributed planar data set.

```
N <- 400  # number of points per class
D <- 2  # dimensionality
K <- 2  # number of classes
X <- data.frame()  # data matrix (each row = single example)
Y <- data.frame()  # class labels

set.seed(308)

for (j in (1:2)) {
    r <- seq(0.05, 1, length.out = N)  # radius
    t <- seq((j - 1) * 4.7, j * 4.7, length.out = N) + rnorm(N,
        sd = 0.3)  # theta
    Xtemp <- data.frame(x = r * sin(t), y = r * cos(t))
    ytemp <- data.frame(matrix(j, N, 1))
    X <- rbind(scale(X), Xtemp)
    Y <- rbind(Y, ytemp)
}

data <- cbind(X, Y)
colnames(data) <- c(colnames(X), "label")

x_min <- min(X[, 1]) - 0.2
x_max <- max(X[, 1]) + 0.2
y_min <- min(X[, 2]) - 0.2
y_max <- max(X[, 2]) + 0.2
```

```
ggplot(data) + geom_point(aes(x = x,
                              y = y,
                              color = as.character(label)),
                          size = 1) +
  theme_bw(base_size = 15) +
  xlim(x_min, x_max) +
  ylim(y_min, y_max) +
  coord_fixed(ratio = 0.8) +
  theme(axis.ticks=element_blank(),
        panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(),
        axis.text=element_blank(),
        axis.title=element_blank(),
        legend.position = 'none')
```

We will split the data set shown in Fig. 4.2, into training and testing data sets (Figs. 4.3 and 4.4).

```
indexes <- sample(1:800, 600)
train_data <- data[indexes, ]
test_data <- data[-indexes, ]
trainX <- train_data[, c(1, 2)]
trainY <- train_data[, 3]
testX <- test_data[, c(1, 2)]
testY <- test_data[, 3]
trainY <- ifelse(trainY == 1, 0, 1)
testY <- ifelse(testY == 1, 0, 1)
```

We need to be sure about the dimensions of the training set data and test set data.

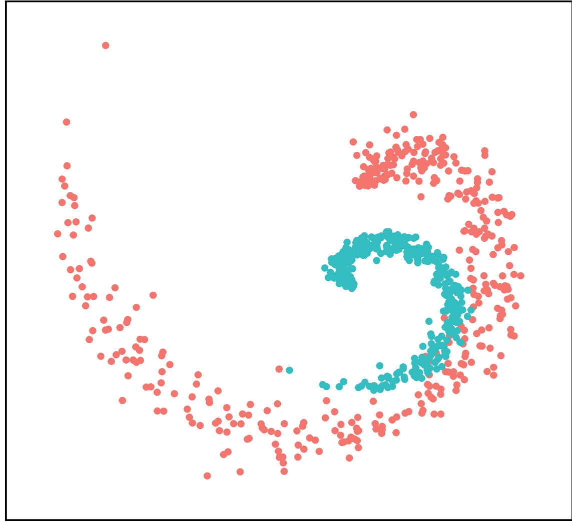


Fig. 4.2 A spiral planar data set created to visualize the effects of different parameter initializations

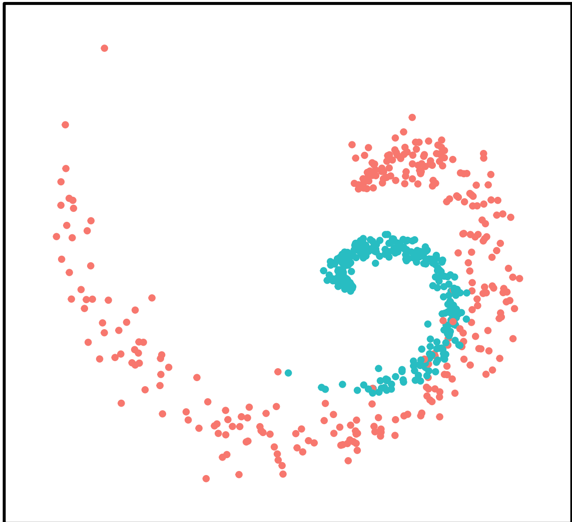


Fig. 4.3 Spiral planar training data set

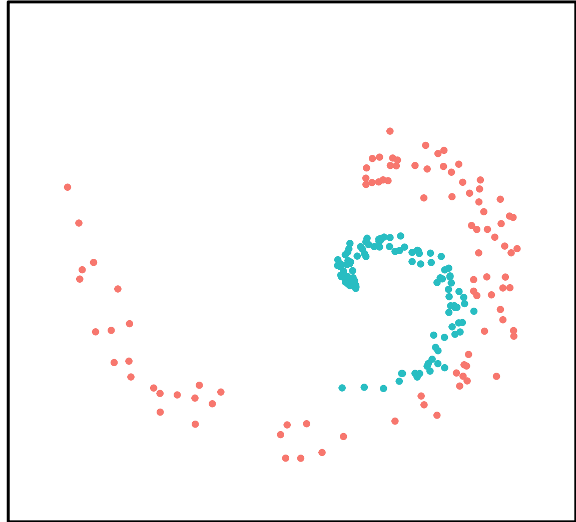
```
dim(train_data)
```

```
[1] 600   3
```

```
dim(test_data)
```

```
[1] 200   3
```

Fig. 4.4 Spiral planar testing data set



4.1.1 Breaking Symmetry

During forward propagation, each unit in the hidden layer gets signal $a_i = \sum_i^n W \cdot x_i$. When we initialize all the weights to the same value, each hidden unit receives the same signal, i.e., if all the weights are initialized to a number n, each unit gets the same signal every time and would not be able to converge during gradient descent.

If all weights are initialized randomly, then each time the model would search for a different path to converge and there would be a better chance to find the global minima. This is what is meant by **breaking symmetry**. The initialization is asymmetric, i.e., it is different and the optimization algorithm will find different solutions to the same problem, thereby seeking a faster convergence.

4.1.2 Zero Initialization

Zero initialization does not serve any purpose because the neural network model does not perform symmetry breaking. If we set all the weights to be zero, then all the neurons of all the layers perform the same calculation, giving the same output. When the weights are zero, the complexity of our network reduces to that of a single neuron.

```
layers_dims <- c(2, 100, 1)

init_zero <- n_layer_model(t(trainX), trainY, t(testX), testY,
    layers_dims, hidden_layer_act = "relu", output_layer_act = "sigmoid",
```

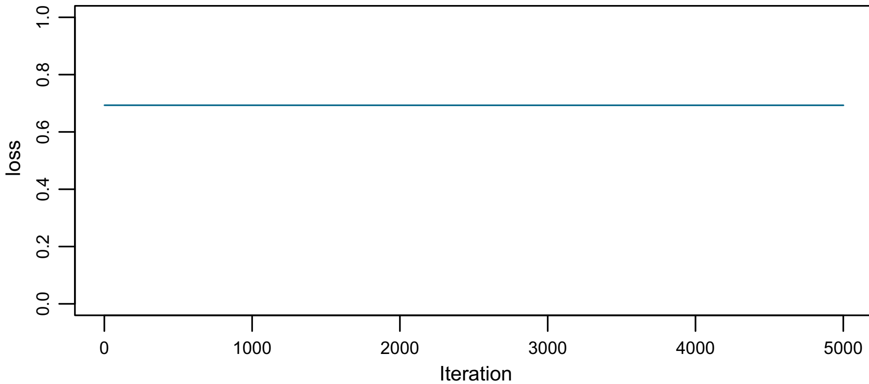


Fig. 4.5 Zero initialization: loss versus iteration

```
    learning_rate = 0.03, num_iter = 5000, initialization = "zero",
    print_cost = T)
```

```
Cost after iteration 0 = 0.693147
Cost after iteration 1000 = 0.693008
Cost after iteration 2000 = 0.693008
Cost after iteration 3000 = 0.693008
Cost after iteration 4000 = 0.693008
Cost after iteration 5000 = 0.693008
Cost after iteration  5000, = 0.693008;
                        Train Acc: 50.833, Test Acc: 47.500,
Application running time: 43.305 minutes
```

From Fig. 4.5 , we can see that zero initialization serves no purpose. The neural network does not break symmetry.

Let us look at the decision boundary for our zero-initialized classifier on the train and test data sets (Fig. 4.6).

```
step <- 0.01

x_min <- min(trainX[, 1]) - 0.2
x_max <- max(trainX[, 1]) + 0.2
y_min <- min(trainX[, 2]) - 0.2
y_max <- max(trainX[, 2]) + 0.2

grid <- as.matrix(expand.grid(seq(x_min, x_max, by = step), seq(y_min,
    y_max, by = step)))
Z <- predict_model(init_zero$parameters, t(grid), hidden_layer_act = "relu",
    output_layer_act = "sigmoid")
Z <- ifelse(Z == 0, 1, 2)

g1 <- ggplot() + geom_tile(aes(x = grid[, 1], y = grid[, 2],
    fill = as.character(Z)), alpha = 0.3, show.legend = F) +
    geom_point(data = train_data, aes(x = x, y = y, color = as.character(trainY)),
```

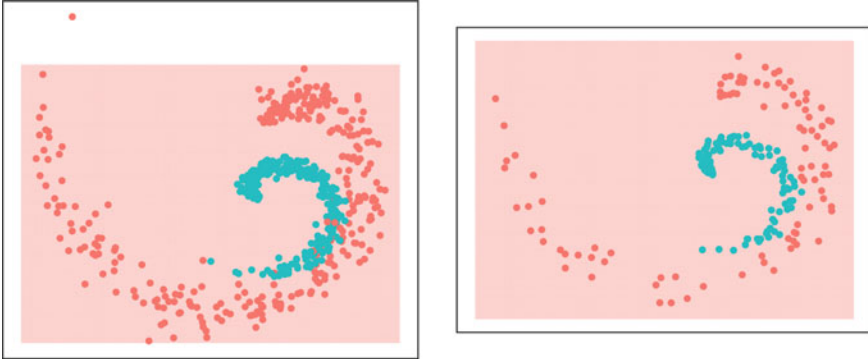


Fig. 4.6 Decision boundary with zero initialization on the training data set (left-hand plot) and the testing data set (right- hand plot)

```
        size = 1) + theme_bw(base_size = 15) + coord_fixed(ratio = 0.8) +
    theme(axis.ticks = element_blank(), panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(), axis.text = element_blank(),
        axis.title = element_blank(), legend.position = "none")

x_min <- min(testX[, 1]) - 0.2
x_max <- max(testX[, 1]) + 0.2
y_min <- min(testX[, 2]) - 0.2
y_max <- max(testX[, 2]) + 0.2

grid <- as.matrix(expand.grid(seq(x_min, x_max, by = step), seq(y_min,
    y_max, by = step)))
Z <- predict_model(init_zero$parameters, t(grid), hidden_layer_act = "relu",
    output_layer_act = "sigmoid")
Z <- ifelse(Z == 0, 1, 2)
g2 <- ggplot() + geom_tile(aes(x = grid[, 1], y = grid[, 2],
    fill = as.character(Z)), alpha = 0.3, show.legend = F) +
    geom_point(data = test_data, aes(x = x, y = y, color = as.character(testY)),
        size = 1) + theme_bw(base_size = 15) + coord_fixed(ratio = 0.8) +
    theme(axis.ticks = element_blank(), panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(), axis.text = element_blank(),
        axis.title = element_blank(), legend.position = "none")

grid.arrange(g1, g2, ncol = 2, nrow = 1)
```

As expected, zero initialization is not able to find any decision boundary.

4.1.3 Random Initialization

One of the ways is to assign the weights from a Gaussian distribution which would have zero mean and some finite variance. This breaks the symmetry and gives better accuracy because now, every neuron is no longer performing the same computation. In this method, the weights are initialized very close to zero.

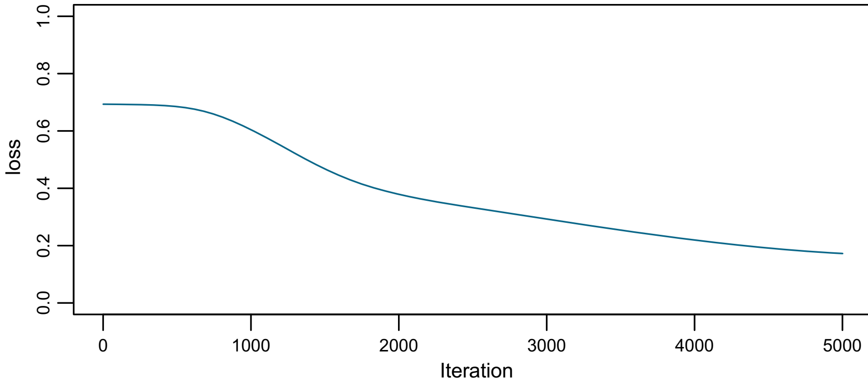


Fig. 4.7 Random initialization loss versus iteration

```
layers_dims <- c(2, 100, 1)

init_random <- n_layer_model(t(trainX),
                             trainY,
                             t(testX),
                             testY,
                             layers_dims,
                             hidden_layer_act = 'relu',
                             output_layer_act  = 'sigmoid',
                             learning_rate = 0.03,
                             num_iter = 5000,
                             initialization = "random",
                             print_cost = T)
```

```
Cost after iteration 0 = 0.693335
Cost after iteration 1000 = 0.603099
Cost after iteration 2000 = 0.378896
Cost after iteration 3000 = 0.293113
Cost after iteration 4000 = 0.218871
Cost after iteration 5000 = 0.172451
Cost after iteration  5000, = 0.172451;
                    Train Acc: 96.333, Test Acc: 97.500,
Application running time: 43.305 minutes
```

The random (Gaussian normal) initialization yields a decreasing cost and returns a training accuracy of 96.3% and a corresponding testing accuracy of 97.5% albeit, it takes 43.3 minutes to converge (Fig. 4.7).

Let us look at the decision boundary and plot our random (Gaussian normal) initialized classifier on the train and test data sets (Fig. 4.8).

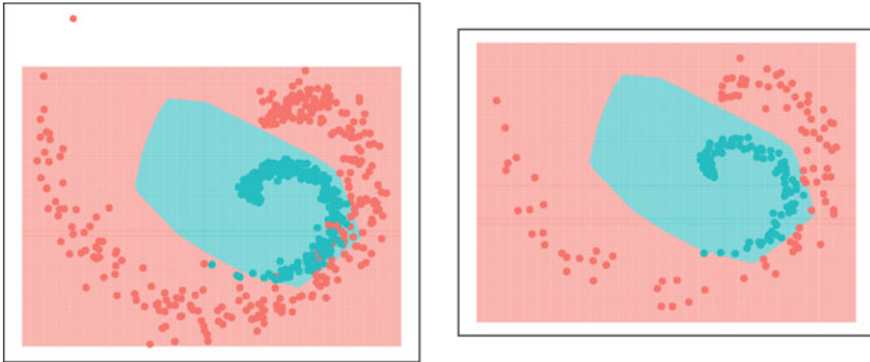


Fig. 4.8 Decision boundary with random initialization on the training data set (left-hand plot) and on the testing data set (right-hand plot)

4.1.4 `Xavier` Initialization

Let us consider a linear neuron represented as $y = w_1x_1 + w_2x_2 + \cdots + w_nx_n$. We would want the variance to remain the same with each passing layer. This helps us to keep the activation values from exploding to a high value or vanishing to zero. We would therefore need to initialize the weights in such a way that the variance remains the same for both x and y, by a process known as `Xavier` initialization. For this purpose, we can write

$$var(y) = var(w_1x_1 + w_2x_2 + \cdots + w_nx_n)$$

The right-hand side of the above equation can be generalized as

$$var(w_ix_i) = E[x_i]^2 var(w_i) + E[w_i]^2 var(x_i) + var(w_i)var(x_i)$$

Considering that the input values and the weight parameter values are coming from a Gaussian distribution with zero mean, the above equation reduces to

$$var(w_ix_i) = var(x_i)var(w_i)$$

or,

$$var(y) = var(w_1)var(x_1) + var(w_2)var(x_2) + \cdots + var(w_n)var(x_n)$$

Since they are all identically distributed, we can represent the above equation as

$$var(y) = n \times var(w_i) \times var(x_i)$$

If the variance of y needs to be the same as that of x, then the term $n \times var(w_i)$ should be equal to 1, or

$$var(w_i) = \frac{1}{n}$$

We therefore need to pick the parameter weights from a Gaussian distribution with zero mean and a variance of $1/n$, where n is the number of input neurons.

In the original paper, the authors [14], take the average of the number input neurons and the output neurons. So the formula becomes

$$var(w_i) = \frac{1}{n_{avg}}$$
$$\text{where,} n_{[avg]} = (n[l-1] + n_{[l]})/2$$
$$w_i = \sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}} \quad (4.1.1)$$

Since it is computationally expensive to implement, we only take the number of input neurons of the previous layer and therefore, Eq. 4.1.1 can be written as

$$w_i = \sqrt{\frac{2}{n^{[l-1]}}} \quad (4.1.2)$$

```
layers_dims <- c(2, 100, 1)

init_Xavier <- n_layer_model(t(trainX),
                             trainY,
                             t(testX),
                             testY,
                             layers_dims,
                             hidden_layer_act = 'relu',
                             output_layer_act  = 'sigmoid',
                             learning_rate = 0.03,
                             num_iter = 5000,
                             initialization = "Xavier",
                             print_cost = T)
```

```
Cost after iteration 0 = 0.911371
Cost after iteration 1000 = 0.237626
Cost after iteration 2000 = 0.157498
Cost after iteration 3000 = 0.126888
Cost after iteration 4000 = 0.111465
Cost after iteration 5000 = 0.102719
Cost after iteration  5000, = 0.102719;
                    Train Acc: 97.667, Test Acc: 99.000,
Application running time: 1.090 minutes
```

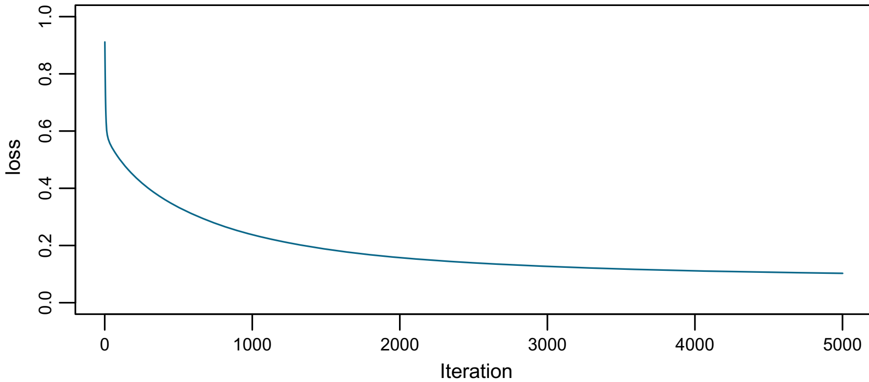


Fig. 4.9 Xavier initialization: loss versus iteration

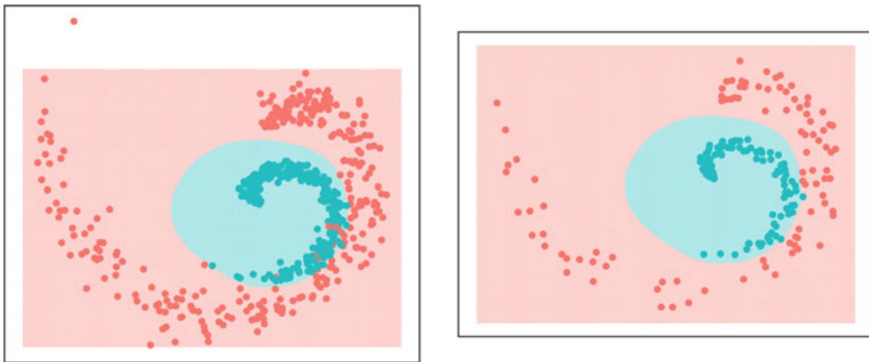


Fig. 4.10 Decision boundary with Xavier initialization on the training data set (left-hand plot) and the testing data set (right- hand plot)

The `Xavier` initialization yields a decreasing cost and returns a higher training accuracy of 97.66% and testing accuracy is 99%. The time to converge has dropped from 55 min to just about a minute (Fig. 4.9). The decision boundary for our data set, using Xavier initialisation is shown in Fig. 4.10.

4.1.5 `He` Initialization

`He` initialization is named after the first author of [15], 2015. The underlying idea behind both, `He` and `Xavier` initialization is to preserve the variance of activation values between layers. In this method, the weights are initialized as a function of the size of the previous layer, which helps in attaining a global minimum of the loss function faster and more efficiently. The weights are still random but differ in range

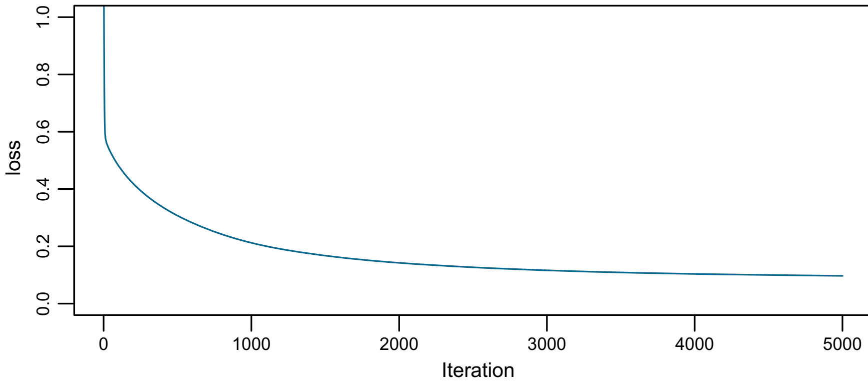


Fig. 4.11 He initialization loss: versus iteration

depending on the size of the previous layer of neurons thus providing a controlled initialization and hence, a faster and more efficient gradient descent. Both the `He` and `Xavier` initialization methods are able to converge faster than random initialization, but with `He` initialization, the errors start to reduce earlier (Fig. 4.11).

```
layers_dims <- c(2, 100, 1)

init_He <- n_layer_model(t(trainX),
                         trainY,
                         t(testX),
                         testY,
                         layers_dims,
                         hidden_layer_act='relu',
                         output_layer_act  = 'sigmoid',
                         learning_rate = 0.03,
                         num_iter = 5000,
                         initialization = "He",
                         print_cost = T)
```

```
Cost after iteration 0 = 1.235932
Cost after iteration 1000 = 0.212091
Cost after iteration 2000 = 0.142321
Cost after iteration 3000 = 0.116169
Cost after iteration 4000 = 0.103672
Cost after iteration 5000 = 0.097017
Cost after iteration  5000, = 0.097017;
                    Train Acc: 97.333, Test Acc: 99.000,
Application running time: 1.038 minutes
```

The `He` initialization yields a steeper decline of the cost and returns a training accuracy of 97.33% with a corresponding testing accuracy of 99%. The decision boundary for our data set, using He initialization is shown in Fig. 4.12. The time to converge is also lower than all other initializations (Fig. 4.13 and Tables 4.1, 4.2).

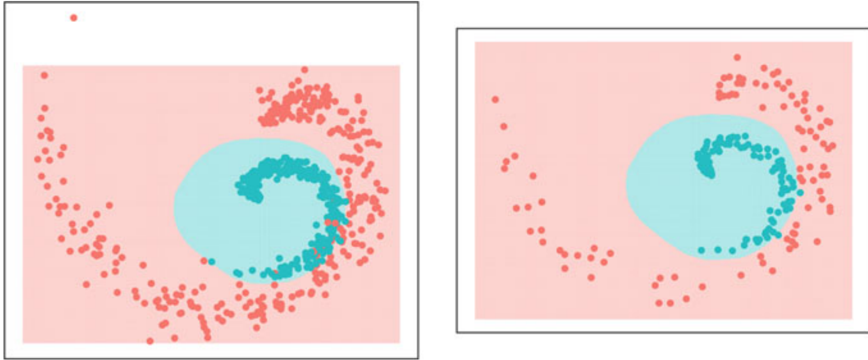


Fig. 4.12 Decision boundary with He initialization on the training data set (left-hand plot) and the testing data set (right- hand plot)

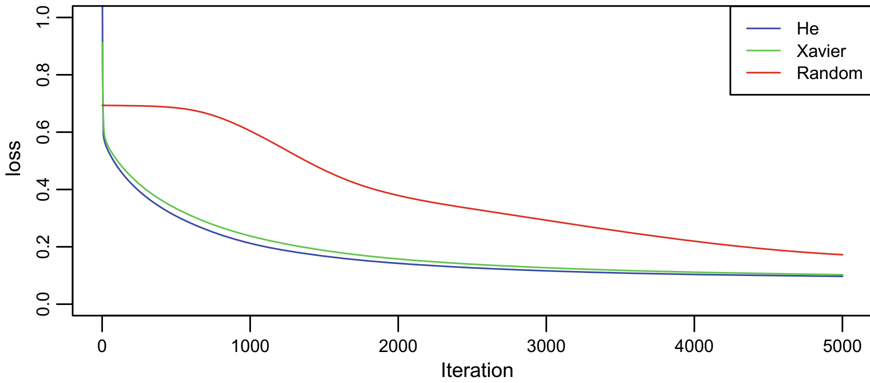


Fig. 4.13 Convergence pattern of different initialization methods

Table 4.1 Time to converge for different initializations

	Random	Xavier	He
Cost	0.172	0.102	0.097
Time (minutes)	43.300	1.090	1.030

There is no specific rule for selecting any specific initialization method, though the `He` initialization works well for networks with `relu` activations. Our learnings are the following:

(a) Different initializations lead to different results.
(b) A well-chosen initialization can speedup the convergence of gradient descent.
(c) A well-chosen initialization method can increase the odds of the gradient descent converging to a lower training (and generalization) error.

Table 4.2 Train and test accuracies with different initializations

Initialization	Train accuracy	Test accuracy
Zero	50.83	47.5
Random	96.30	97.5
Xavier	97.60	99.0
He	97.30	99.0

(d) The parameter weights $W^{[l]}$ should be initialized randomly to break the symmetry and makes sure that different hidden units can learn different things.
(e) We should not initialize the parameter weights to large values.
(f) `He` initialization works best for networks with `relu` activations.
(g) It is appropriate to initialize the biases $b^{[l]}$ to zeros. Symmetry is still broken so long as $W^{[l]}$ is initialized randomly.

4.2 Dealing with NaNs

Having a model which yields NaNs or Infs is quite common if some of the components in the model are not set properly. NaNs are hard to deal with because, it may be caused by a bug or an error in the code or because of the numerical stability in the computational environment (including libraries, etc.). In some cases, it could relate to the algorithm itself. Let us outline some of the common issues which can cause the model to yield NaNs, and some of the ways to get around this problem.

4.2.1 Hyperparameters and Weight Initialization

Most frequently, the cause would be that some of the hyperparameters, especially learning rates, are set incorrectly. A high learning rate can result in NaN outputs so the first and easiest solution is to lower it. One suggested method is to keep halving the learning rate till the NaNs disappear.

The penalty (λ) in the regularization term can also play a part where the model throws up NaN values. Using a wider hyperparameter space with one or two training epochs, each could be tried out to see if the NaNs disappear.

Some models can be very sensitive to the weight initialization. If the weights are not initialized correctly, the model can end up yielding NaNs.

4.2.2 Normalization

Sometimes, this may be obviated by normalizing the input values (though, normalization is a norm which must be strictly followed).

4.2.3 Using Different Activation Functions

Try using other activation functions like tanh. Unlike ReLUs, the outputs from tanh have an upper bound in value and may be a solution. Adding more nonlinearity can also help.

4.2.4 Use of NanGuardMode, DebugMode, or MonitorMode

If adjusting the hyperparameters do not work help, it can be still be sought from Theano's `NanGuardMode`, by changing the mode of the Theano function. This will monitor all input/output variables in each node, and raise an error if NaNs are detected. Similarly, Theano's DebugMode and MonitorMode can also help.

4.2.5 Numerical Stability

This may happen due to zero division or by any operation that is making a number(s) extremely large. Some functions like, $\frac{1}{log(p(x)+1)}$ could result in NaNs for those nodes, which have learned to yield a low probability $p(x)$ for some input x. It is important to find what are the function input values for the given cost (or any other) function are and why we are getting that input. Scaling the input data, weight initialization, and using an adaptive learning rate is some of the suggested solutions.

4.2.6 Algorithm Related

If the above methods fail, there could be a good chance that something has gone wrong in the algorithm. In that case, we need to inspect the mathematics in the algorithm and try to find out if everything has been derived correctly.

4.2.7 NaN Introduced by AllocEmpty

AllocEmpty is used by many operations such as scan to allocate some memory without properly clearing it. The reason for that is that the allocated memory will subsequently be overwritten. However, this can sometimes introduce NaN depending on the operation and what was previously stored in the memory it is working on. For instance, trying to zero out memory using a multiplication before applying an operation could cause a NaN if NaN is already present in the memory, since $0 * \text{NaN} \rightarrow \text{NaN}$.

4.3 Conclusion

We have explored different initialization techniques used in neural networks and learnt how they affect both convergence time and accuracy. In deep neural networks where it takes a long time to train a model, initialization of the parameters makes a big difference.

We have also learnt the hazards of encountering NaN values and how to counter them.

In the next chapter, we will discuss different optimization techniques which will further enhance the performance of the deep neural network models.