README.md

2024-02-14

```r
knitr::opts_chunk$set(echo = TRUE)

relu <- function(x) {
  # Rectified linear unit activation function
  # Replaces negative values with zero
  x[x < 0] <- 0
  x
}

softmax <- function(x) {
  # Softmax activation function
  # Normalizes the input to a probability distribution
  exp_x <- exp(x)
  row_sums <- apply(exp_x, 1, sum)
  exp_x / row_sums
}

one_hot_encode <- function(x) {
  # One-hot encoding function
  # Converts a vector of labels into a matrix of binary indicators
  n <- length(x)
  k <- max(x)
  mat <- matrix(0, nrow = n, ncol = k)
  mat[cbind(1:n, x)] <- 1
  mat
}

    # Initialize the parameters with random values
initialize <- function(n_hidden, n_features = 2, n_class = 2) {
      return(list(
        W1 = matrix(rnorm(n_features * n_hidden), nrow = n_features, ncol = n_hidden),
        b1 = rnorm(n_hidden),
        W2 = matrix(rnorm(n_hidden * n_class), nrow = n_hidden, ncol = n_class),
        b2 = rnorm(n_class)
      ))
}


    # forwardPropogate pass of the neural network
forwardPropogate <-  function( input_data,params) {
      W1 <- params$W1
      W2 <- params$W2
      b1 <- params$b1
      b2 <- params$b2
    a1 <-  (input_data %*% W1) + b1
    params$z1 <- relu(a1)
    a2 <- (params$z1 %*% W2) + b2
    params$z2 <- softmax(a2)

    return(list(params$z1, params$z2))
  }

  # Fit the neural network to the input data and labels
fit = function(input_data, label, batch_size, iter_num, params) {
    for (epoch in 1:iter_num) { #Iterate through the requested number of iterations
      p <- sample(1:length(label)) # Use sample to generate a random int
      input_data <- input_data[p, ] # select the row
      label <- label[p] #select the label
      for (i in seq(1, length(label), by = batch_size)) {
        batch_data <- input_data[i:(i + batch_size - 1), ]
        batch_label <- label[i:(i + batch_size - 1)]
        params <- sgd(batch_data, batch_label, params) # We update the params
      }
    }
  return(params)
  }

  # Stochastic gradient descent update of the parameters
  sgd <- function(data, label, params, alpha = 1e-4) {
    grad <- backward(data, label, params)
    for (layer in names(grad)) {
      params[[layer]] <- params[[layer]] + (alpha * grad[[layer]])
    }
    return(params)
  }

  # Backward pass of the neural network
backward <- function(data, label, params) {
    W1 <- params$W1
    W2 <- params$W2
    b1 <- params$b1
    b2 <- params$b2
    z1 <- forwardPropogate(data, params)[[1]]
    z2 <- forwardPropogate(data, params)[[2]]

    #label <- one_hot_encode(label)
    db2_temp <- label - z2
    db2 <- colSums(db2_temp)
    dW2 <- t(z1) %*% db2_temp
    db1_temp <- db2_temp %*% t(W2)
    db1_temp[z1 <= 0] <- 0
    db1 <- colSums(db1_temp)
    dW1 <- t(data) %*% db1_temp

    return(list(W1 = dW1, b1 = db1, W2 = dW2, b2 = db2))
  }
```

```
    # Test the accuracy of the neural network on the test data and labels
  test = function(train_data, train_label, test_data, test_label, batch_size, iter_num, params) {
      pred_label <- forwardPropogate(test_data, params)[[2]]
      pred_label <- apply(pred_label, 1, which.max)
      acc <- mean(pred_label == test_label)
      return(acc)
    }
```

```
set.seed(123)
n_train <- 100
n_test <- 20
n_features <- 2
n_class <- 2
train_data <- matrix(rnorm(n_train * n_features), nrow = n_train, ncol = n_features)
train_label <- sample(1:n_class, n_train, replace = TRUE)
test_data <- matrix(rnorm(n_test * n_features), nrow = n_test, ncol = n_features)
test_label <- sample(1:n_class, n_test, replace = TRUE)

# Instantiate the NeuralNet class with 10 hidden units
nn <- initialize(n_hidden = 10)

# Train the neural network with batch size of 10 and 50 iterations
nn <- fit(train_data, train_label, batch_size = 10, iter_num = 1, nn)

# Test the accuracy of the neural network on the test data
acc <- test(train_data, train_label, test_data, test_label, batch_size = 10, iter_num = 50, nn)
cat("Accuracy:", acc, "\n")
```

```
## Accuracy: 0.5
```

```
knitr::opts_chunk$set(echo = TRUE)

# The neural network so to speak is just weights and biases
set.seed(123)


relu <- function(x) {
  # Rectified linear unit activation function
  # Replaces negative values with zero
  x[x < 0] <- 0
  x
}

softmax <- function(x) {
  # Softmax activation function
  # Normalizes the input to a probability distribution
  exp_x <- exp(x)
  row_sums <- apply(exp_x, 1, sum)
  exp_x / row_sums
```

```r
}

one_hot_encode <- function(x) {
  # One-hot encoding function
  # Converts a vector of labels into a matrix of binary indicators
  n <- length(x)
  #k <- max(x)
  k <- length(unique(x))
  mat <- matrix(0, nrow = n, ncol = k)
  mat[cbind(1:n, x)] <- 1
  mat
}

one_hot_encode <- function(x) {
  # One-hot encoding function
  # Converts a vector of string labels into a matrix of binary indicators
  n <- length(x)
  k <- length(unique(x))
  labels <- unique(x)
  mat <- matrix(0, nrow = n, ncol = k)
  for (i in 1:n) {
    j <- which(labels == x[i])
    mat[i, j] <- 1
  }
  mat
}


string_to_int <- function(x) {
  unique_strings <- unique(x)
  int_values <- seq_along(unique_strings)
  x_int <- int_values[match(x, unique_strings)]
  return(x_int)
}



n_train <- 100
n_test <- 20
n_features <- 4
n_class <- 3

## generate a random ordering
set.seed(1) ## make reproducible here, but not if generating many random samples
rand <- sample(nrow(iris))
rand
```

```
##   [1]  68 129  43  14  51  85  21 106  74   7  73  79  37 105 110  34 143 126
##  [19]  89  33  84  70 142  42  38 111  20  28 124  44  87 149  40 121  25 119
##  [37]  39 146 127   6  24  32 147   2  45  18  22  78 102  65 115 120 100  75
##  [55]  81  13 118 132  48  93  23 130  29  95 104 123  92 131 134 144  31  17
##  [73] 140  91  64  60 113 135  10   1 148  59  26  15  58  88 136 112  77  53
##  [91]  12 114  76  61 145  86  94  83  19 150  35  98  71 101 108  55 125  56
## [109]  41 138   3  82  50 141 133  27  63 122 116  66  52  96  16  30   4 107
## [127]  49  62  97 103   9  99 109 137  54  90 139  11  80  69  36   8  67  46
## [145] 128  47 117   5  57  72
```

```
d <- iris[rand,]
train_data <- as.matrix(d[0:75,0:4])

train_label <- one_hot_encode(as.matrix(d[0:75,5:5]))


test_data <- as.matrix(d[75:150,0:4])

test_label <- one_hot_encode(as.matrix(d[75:150,5:5]))

n_hidden = 10

params = list(
        W1 = matrix(rnorm(n_features * n_hidden), nrow = n_features, ncol = n_hidden),
        b1 = rnorm(n_hidden),
        W2 = matrix(rnorm(n_hidden * n_class), nrow = n_hidden, ncol = n_class),
        b2 = rnorm(n_class)
      )

iter_num = 10
batch_size = 10
alpha = 1e-4

    # Backward pass of the neural network
  backward <- function(data, label, params) {
      W1 <- params$W1
      W2 <- params$W2
      b1 <- params$b1
      b2 <- params$b2
      z1 <- forwardPropogate(data, params)[[1]]
      z2 <- forwardPropogate(data, params)[[2]]

      db2_temp <- label - z2
      db2 <- colSums(db2_temp)
      dW2 <- t(z1) %*% db2_temp
      db1_temp <- db2_temp %*% t(W2)
      db1_temp[z1 <= 0] <- 0
      db1 <- colSums(db1_temp)
      dW1 <- t(data) %*% db1_temp

      return(list(W1 = dW1, b1 = db1, W2 = dW2, b2 = db2))
  }

  forwardPropogate <-  function( input_data,params) {
      W1 <- params$W1
      W2 <- params$W2
      b1 <- params$b1
      b2 <- params$b2

      a1 <-  (input_data %*% W1) + b1
      params$z1 <- relu(a1)
      a2 <- (params$z1 %*% W2) + b2
      params$z2 <- softmax(a2)

      return(list(params$z1, params$z2))
    }

  train_dim <- dim(train_label)[1]

s <- seq(1, train_dim, by = batch_size)

y <- pmin(s+batch_size, train_dim)
l <- seq(1, length(y))

      for (epoch in 1:iter_num) { #Iterate through the requested number of iterations
        p <- sample(1:train_dim) # Use sample to generate a random int
        input_data <- train_data[p, ] # select the row
        label <- train_label[p] #select the label
        for (k in l) {
          i <- s[k]
          j <- y[k]

          batch_data <- train_data[i:j, ]

          batch_label <- train_label[i:j]

           #   sgd <- function(data, label, params, alpha = 1e-4) {

      grad <- backward(batch_data, batch_label, params)



      for (layer in names(grad)) {
        params[[layer]] <- params[[layer]] + (alpha * grad[[layer]])
      }


         # params <- sgd(batch_data, batch_label, params) # We update the params
        }

        }

    # Test the accuracy of the neural network on the test data and labels
  test = function(train_data, train_label, test_data, test_label, params) {
      pred_label <- forwardPropogate(test_data, params)[[2]]
      pred_label <- apply(pred_label, 1, which.max)
      acc <- mean(pred_label == test_label)
      return(acc)
    }



# Test the accuracy of the neural network on the test data
```

```r
acc <- test(train_data, train_label, test_data, test_label,params)
cat("Accuracy:", acc, "\n")
```

```
## Accuracy: 0.1885965
```

From a mathematical perspective, the gradient computation can be understood using the chain rule of calculus and the definition of the loss function.

Let's denote the loss function as `L`, which is a function of the predicted output `z2` and the true labels `label`. The goal is to compute the gradient of the loss function with respect to the weights of the second layer `W2`, denoted as

$$\frac{\partial L}{\partial W_2}$$

.

Using the chain rule, we can write the gradient of the loss function with respect to `W2` as:

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2}$$

Here,

$$\frac{\partial L}{\partial z_2}$$

represents the gradient of the loss function with respect to the predicted output `z2`, and

$$\frac{\partial z_2}{\partial W_2}$$

represents the gradient of the predicted output `z2` with respect to the weights `W2`.

Now, let's assume the loss function is a mean squared error (MSE) between the predicted output `z2` and the true labels `label`, which is a common choice in neural networks:

$$L = \frac{1}{2}\left(\text{label} - z_2\right)^2$$

Using the chain rule, we can compute the gradient of the loss function with respect to `z2` as:

$$\frac{\partial L}{\partial z_2} = -\left(\text{label} - z_2\right)$$

This is because derivative of the squared error with respect to `z2` is

$$-2 \cdot \left(\text{label} - z_2\right)$$

, and we divide by 2 to get the average error.

Now, we need to compute the gradient of the predicted output `z2` with respect to the weights `W2`. Assuming the neural network has a linear activation function in the second layer, we can write `z2` as:

$$z_2 = W_2 \cdot z_1 + b_2$$

where `z1` is the output of the first layer, and `b2` is the bias term in the second layer.

Using the product rule, we can compute the gradient of `z2` with respect to `W2` as:

$$\frac{\partial z_2}{\partial W_2} = z_1$$

This is because the derivative of

$$W_2 \cdot z_1$$

with respect to `W2` is `z1`, and the derivative of `b2` with respect to `W2` is 0.

Now, we can plug in the expressions for

$$\frac{\partial L}{\partial z_2}$$

and

$$\frac{\partial z_2}{\partial W_2}$$

into the chain rule formula:

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2} = -\left(\text{label} - z_2\right) \cdot z_1$$

This is exactly what the line of code `dW2 <- t(z1) %*% db2_temp` is computing! The transpose of `z1` is used to ensure the correct matrix multiplication, and `db2_temp` represents the error term `label - z2`.

By computing the gradient of the loss function with respect to the weights `W2` in this way, we can update the weights using an optimization algorithm such as stochastic gradient descent (SGD) to minimize the loss function.

The chain rule is a fundamental concept in calculus that allows us to compute the derivative of a composite function. In this case, we have a composite function:

$$L = L(z_2)$$

where L is the loss function, and z_2 is the output of the second layer.

The chain rule states that if we have a composite function $f(g(x))$, where f and g are both functions of x, then the derivative of f with respect to x is given by:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x}$$

In our case, we can apply the chain rule by identifying the inner function g as z_2, and the outer function f as L. Specifically, we can write:

$$L = L(z_2(W_2, z_1, b_2))$$

where z_2 is a function of W_2, z_1, and b_2.

Now, we want to compute the derivative of L with respect to W_2. Using the chain rule, we can write:

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2}$$

Here, $\frac{\partial L}{\partial z_2}$ is the derivative of the loss function L with respect to the output z_2, and $\frac{\partial z_2}{\partial W_2}$ is the derivative of the output z_2 with respect to the weights W_2.

The key insight is that we can separate the computation of the derivativo parts:

1. Compute the derivative of the loss function L with respect to the output z_2. This is a simple computation, since L is a function of z_2 only.

$$\frac{L}{\partial z_2} = -(y - z_2)$$

where y is the true label.

2. Compute the derivative of the output z_2 with respect to the weights W_2. This is also a simple computation, since z_2 is a function of W_2, z_1, and b_2.

$$\frac{\partial z_2}{\partial W_2} = z_1$$

By multiplying these two derivatives together, we get the final result:

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2} = -(y - z_2) \cdot z_1$$

This is exactly what the line of code `dW2 <- t(z1) %*% db2_temp` is computing! The transpose of z_1 is used to ensure the correct matrix multiplication, and `db2_temp` represents the error term $y - z_2$.

Here is the rewritten explanation with LaTeX expressions:

When computing the derivative of the loss function with respect to the weights W_2, we can ignore the inputs z_1 and b_2 because they are not directly affected by the weights W_2.

To see why, let's revisit the expression for the output z_2:

$z_2 = W_2 \cdot z_1 + b_2$

When computing the derivative of z_2 with respect to W_2, we only care about how W_2 affects the output z_2. The inputs z_1 and b_2 are treated as constants, because they are not functions of W_2.

In other words, when we compute the derivative of z_2 with respect to W_2, we are asking: "How does W_2 change the output z_2, assuming z_1 and b_2 are fixed?"

Using the product rule, we can compute the derivative of z_2 with respect to W_2 as:

$\frac{\partial z_2}{\partial W_2} = z_1$

Here, we've ignored the contribution of b_2 because it's a constant term that doesn't depend on W_2. Similarly, we've treated z_1 as a constant, because it's an input that's not directly affected by W_2.

By ignoring z_1 and b_2 in this way, we're effectively assuming that they are fixed inputs that don't change when we update the weights W_2. This is a reasonable assumption, because we're only interested in how the weights W_2 affect the output z_2, not how the inputs z_1 and b_2 affect the output.

Of course, this assumption is only valid if the inputs z_1 and b_2 are indeed fixed and don't depend on the weights W_2. In general, if the inputs do depend on the weights, we would need to take that into account when computing the derivative.

But in this specific case, ignoring $_1$ and b_2 allows us to focus on the direct effect of W_2 on the output z_2, which is exactly what we need to compute the gradient of the loss function with respect to the weights.