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 + .04)
}

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
}


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
}

    # Initialize the parameters with random values
initialize <- function(n_hidden, n_features, n_class) {
```

```r
      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
      z1 <- relu(a1)
      a2 <- (z1 %*% W2) + b2
      z2 <- softmax(a2)

      return(list(z1,z2))
    }

    # Fit the neural network to the input data and labels
  fit = function(input_data, input_label, batch_size, iter_num=10, params, alpha) {
    #input_data<- test_data2
    #input_label <-test_label2
    #iter_num <-10
    #batch_size <- 10
    #params <- nn
    #alpha <-.04
    label_dimensions <- dim(input_label)[[1]]
      for (epoch in 1:iter_num) { #Iterate through the requested number of iterations
        p <- sample(1:dim(input_label)[1] ) # Use sample to generate a random int
        r <- input_data[p, ] # select the row
        l <- input_label[p] #select the label
        s <- seq(1, label_dimensions, by = batch_size)
        s[length(s)]<- label_dimensions-batch_size
        for (i in s) {
          batch_data <- r[i:(i + batch_size - 1), ]
          batch_label <- l[i:(i + batch_size - 1)]
          params <- sgd(batch_data, batch_label, params, alpha) # 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)) {
```

```r
        params[[layer]] <- params[[layer]] + (alpha * grad[[layer]])
      }
      return(params)
    }

    # Backward pass of the neural network
  backward <- function(batch_data, label, nn) {
      W1 <- nn$W1
      W2 <- nn$W2
      b1 <- nn$b1
      b2 <- nn$b2
      z <- forwardPropogate(batch_data, nn)
      z1 <-z[[1]]
      z2 <- z[[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(batch_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(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)
    }




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)


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

```r
d <- iris[rand,]
train_data2 <- as.matrix(d[0:75,0:4])

train_label2 <- one_hot_encode(as.matrix(d[0:75,5:5]))


test_data2 <- as.matrix(d[75:150,0:4])

test_label2 <- one_hot_encode(as.matrix(d[75:150,5:5]))

# Instantiate the NeuralNet class with 10 hidden units
nn <- initialize(n_hidden = 1, 4, 3)

# Train the neural network with batch size of 10 and 50 iterations
nn <- fit(train_data2, train_label2, batch_size = 75, iter_num = 500, nn, alpha=.4)

# Test the accuracy of the neural network on the test data
acc <- test( test_data2, test_label2,  nn)
cat("Accuracy:", acc, "\n")
```

```
## Accuracy: 0.2236842
```

```r
#set.seed(1) ## make reproducible here, but not if generating many random samples
cross_entropy_loss <- function(y, z) {
  # Compute the softmax function
  p <- exp(z) / sum(exp(z))

  # Compute the error term
  e <- y - p

  # Compute the derivative of the cross-entropy loss with respect to the probabilities
  dL_dp <- - (y / p) + (1 - y) / (1 - p)

  # Compute the derivative of the probabilities with respect to the logits
  dP_dz <- p * (1 - p)

  # Compute the gradient of the cross-entropy loss with respect to the logits
  dL_dz <- dL_dp * dP_dz

  # Compute the cross-entropy loss
  loss <- sum(-y * log(p) - (1 - y) * log(1 - p))

  return(list(loss = loss, dL_dz = dL_dz))
}

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
}

rand <- sample(nrow(iris))
d <- iris[rand,]



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

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




relu <- function(x) {
  # Rectified linear unit activation function
  # Replaces negative values with zero
  x[x < 0] <- 0
  x
}
sgn <- function(x) {
  x[x>0] <- 1
  x[x!=1] <-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 + .04)
}


initialize_neural_network <- function(n_hidden, n_features, n_class) {
      return(list(
        W = t(matrix(rnorm(n_features * n_hidden), nrow = n_features, ncol = n_hidden)),
        b1 = rnorm(n_hidden),
        U = t(matrix(rnorm(n_hidden * n_class), nrow = n_hidden, ncol = n_class)),
        b2 = rnorm(n_class)
      ))
}

# Instantiate the NeuralNet class with 10 hidden units
nn <- initialize_neural_network(n_hidden = 2, 4, n_class =   dim(test_label2)[[2]])
alpha <-1

for (i in 1: 2)
```

```
{
W <- nn$W
U <- nn$U
b1 <- nn$b1
b2 <- nn$b2


## Inputs:
## x = input
## z = Wx+b1
## h = relu(z)
## th = Uh +b2
## yh = softmax(theta)
## J = CE(y, yh)

## Required Gradients
## dj/U ( derivative of cross entropy with respect to the second layer) => dj/dyh * dyh/dth * dth/dU
## Dj/db2 ( derivative of cross entropy with respect to b2)
## Dj/W ( derivative of cross entropy with respect to the first layer)
## Dj/b1 ( derivative of cross entropy with respect to b1)
## Dj/x ( derivative of cross entropy with respect to the inputs)


## Intermediate gradients

## d1 = Dj/dth
## d2 = dj/dz


## forward pass
z <-  W%*%t(x) + b1
h <- relu(z)
th <- (U %*%h) + b2
yh <- softmax(th)

d1 <- t(yh - t(y))
db2 <- colSums(d1)

dU <- t(h %*% d1)
d_yh <- yh - t(y)
d_th <- d_yh * (yh * (1 - yh))
d_h <- t(U) %*% d_th
d_z <- d_h * sgn(z)
d_W <- d_z %*% x
d_b1 <- rowSums(d_z)
d_U <- d_th %*% t(h)
d_b2 <- rowSums(d_th)


grad <-list(W = d_W, b1 = d_b1, U = d_U, b2 = d_b2)

      for (layer in names(grad)) {
        nn[[layer]] <- nn[[layer]] - (alpha * grad[[layer]])
```

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

      pred_label <- apply(t(yh), 1, which.max)
      acc <- mean(pred_label == apply(y, 1, which.max))
cat("Accuracy:", acc, "\n")
}# Forward pass of  the neural network
```

```
## Accuracy: 0.3684211
## Accuracy: 0.3421053
```

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:

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

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 $y - z_2$.

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.

Step 1: Compute the partial derivative of the loss with respect to yh

The partial derivative of the loss with respect to yh is computed as follows:

$$\frac{\partial L}{\partial yh} = \frac{\partial}{\partial yh}(yh - t(y))^2 = 2(yh - t(y))$$

Step 2: Compute the partial derivative of the loss with respect to th

Using the chain rule, we can compute the partial derivative of the loss with respect to th as follows:

$$\frac{\partial L}{\partial th} = \frac{\partial L}{\partial yh}\frac{\partial yh}{\partial th} = 2(yh - t(y))\frac{\partial}{\partial th}softmax(th)$$

Step 3: Compute the partial derivative of the softmax function with respect to th

The partial derivative of the softmax function with respect to th is computed as follows:

$$\frac{\partial}{\partial th}softmax(th) = softmax(th)(1 - softmax(th))$$

Step 4: Compute the partial derivative of the loss with respect to th

Substituting the result from Step 3 into the result from Step 2, we get:

$$\frac{\partial L}{\partial th} = 2(yh - t(y))softmax(th)(1 - softmax(th))$$

Step 5: Compute the partial derivative of the loss with respect to h

Using the chain rule, we can compute the partial derivative of the loss with respect to h as follows:

$$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial th}\frac{\partial th}{\partial h} = 2(yh - t(y))softmax(th)(1 - softmax(th))U$$

Step 6: Compute the partial derivative of the loss with respect to z

Using the chain rule, we can compute the partial derivative of the loss with respect to z as follows:

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial z} = 2(yh - t(y))softmax(th)(1 - softmax(th))Usgn(z)$$

Step 7: Compute the partial derivative of the loss with respect to W

Using the chain rule, we can compute the partial derivative of the loss with respect to W as follows:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial W} = 2(yh - t(y))softmax(th)(1 - softmax(th))Usgn(z)x$$

Step 8: Compute the partial derivative of the loss with respect to b1

Using the chain rule, we can compute the partial derivative of the loss with respect to b1 as follows:

$$\frac{\partial L}{\partial b1} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial b1} = 2(yh - t(y))softmax(th)(1 - softmax(th))Usgn(z)$$

Step 9: Compute the partial derivative of the loss with respect to U

Using the chain rule, we can compute the partial derivative of the loss with respect to U as follows:

$$\frac{\partial L}{\partial U} = \frac{\partial L}{\partial th}\frac{\partial th}{\partial U} = 2(yh - t(y))softmax(th)(1 - softmax(th))h$$

Step 10: Compute the partial derivative of the loss with respect to b2

Using the chain rule, we can compute the partial derivative of the loss with respect to b2 as follows:

$$\frac{\partial L}{\partial b2} = \frac{\partial L}{\partial th}\frac{\partial th}{\partial b2} = 2(yh - t(y))softmax(th)(1 - softmax(th))$$