Irisk Lab report2

Ruibo Hou

2024-02-15

Softmax regression - the final layer of nerual network

1. Data Preprocessing

```r
# Load the data
mnist <- read.csv("https://pjreddie.com/media/files/mnist_train.csv", nrows = 20000)
colnames(mnist) <- c("Digit", paste("Pixel", seq(1:784), sep = ""))

# Normalize the pixel values to range [0, 1]
mnist[, -1] <- mnist[, -1] / 255

# Split the dataset into features (X) and labels (y)
X <- as.matrix(mnist[, -1])
y <- mnist$Digit
```

Explanation: Preprocessing is crucial for machine learning models. Normalizing the pixel values helps in speeding up the convergence during training by ensuring that all features (in this case, pixels) have similar scales. Here, we divide by 255 to scale the pixel values to a [0, 1] range. We also separate the dataset into features (X) and labels (y), which will be used for training the softmax regression model.

2. Initializing Parameters

```r
# Initialize weights and biases
num_classes <- length(unique(y))
num_features <- ncol(X)
weights <- matrix(runif(num_features * num_classes) - 0.5, nrow = num_features, ncol = num_classes)
biases <- runif(num_classes) - 0.5
```

Explanation: Initialization plays a role in the optimization landscape that the training process will navigate. Here, weights and biases are initialized randomly, providing a starting point for optimization. The weight matrix dimensions allow for the calculation of class scores for each of the 10 digits given an input image. Each column in the weight matrix corresponds to a specific class (digit), and biases are used to adjust the output scores independently of the input features.

3. Softmax Function

```r
softmax <- function(scores) {
  exp_scores <- exp(scores)
  probs <- sweep(exp_scores, 1, rowSums(exp_scores), '/')
  return(probs)
}
```

Explanation: The softmax function ensures that the output values are interpretable as probabilities, as they are positive and sum up to 1 for each input. This function is essential for multi-class classification tasks like MNIST, where each output corresponds to the model's confidence in each class. Applying softmax allows us to use cross-entropy as a loss function, which measures the difference between the predicted probabilities and the actual distribution of the labels.

4. Cross-Entropy Loss

```
cross_entropy_loss <- function(probs, y) {
  # Convert labels to one-hot encoding
  y_one_hot <- matrix(0, nrow = nrow(probs), ncol = num_classes)
  y_one_hot[cbind(1:nrow(probs), y + 1)] <- 1  # R is 1-indexed

  # Compute the loss
  loss <- -sum(y_one_hot * log(probs)) / nrow(probs)
  return(loss)
}
```

Explanation: Cross-entropy loss is a key component in training classification models, as it provides a measure of how different the predicted probabilities are from the actual labels. In this implementation, we first convert the labels into a one-hot encoded format, where each label is represented as a vector of zeros except for a single one at the index corresponding to the class label. The loss is calculated by taking the negative log of the predicted probabilities for the actual classes and averaging over all examples. This loss function encourages the model to assign high probabilities to the correct classes.

5. Forward Propagation and Class Score Calculation

```
forward_propagation <- function(X, weights, biases) {
  scores <- X %*% weights + matrix(rep(biases, each=nrow(X)), nrow=nrow(X), ncol=ncol(weights), byrow=T
  probs <- softmax(scores)
  return(list(scores=scores, probs=probs))
}
```

Explanation: Forward propagation calculates the weighted sum of inputs plus biases for each class, which are the raw class scores. The softmax function is then applied to these scores to derive probabilities. Each probability indicates the likelihood of an input belonging to a particular class. This step is crucial for both training (to compute the loss and gradients) and making predictions.

6. Computing the Gradient

```
compute_gradients <- function(X, y, probs, num_classes) {
  # Convert labels to one-hot encoding
  y_one_hot <- matrix(0, nrow = nrow(probs), ncol = num_classes)
  y_one_hot[cbind(1:nrow(probs), y + 1)] <- 1

  # Compute gradients
  dW <- t(X) %*% (probs - y_one_hot) / nrow(X)
  dB <- colSums(probs - y_one_hot) / nrow(X)

  return(list(dW=dW, dB=dB))
}
```

Explanation: The gradient computation is a critical step in training neural networks. Here, dW represents the gradient of the loss with respect to the weights, and dB represents the gradient with respect to the biases. These gradients indicate how much a change in each parameter would affect the loss. By subtracting a portion of these gradients from the parameters, we can iteratively reduce the loss, making our model more accurate

7. Parameter Update (Gradient Descent)

```
update_parameters <- function(weights, biases, dW, dB, learning_rate) {
  weights <- weights - learning_rate * dW
  biases <- biases - learning_rate * dB
  return(list(weights=weights, biases=biases))
}
```

Explanation: This function updates the model parameters in the direction that reduces the loss, as indicated by the gradients. The learning rate controls how big a step we take during each update. If the learning rate is too high, we might overshoot the minimum; if it's too low, training might take too long.

8. Model Evaluation

```
evaluate_accuracy <- function(probs, y) {
  predictions <- max.col(probs) - 1
  accuracy <- sum(predictions == y) / length(y)
  return(accuracy)
}
```

Explanation: After training, evaluating the model's performance on unseen data is crucial. Accuracy measures the proportion of correctly predicted instances. It provides a straightforward metric to gauge how well our model generalizes from the training data to new, unseen data.

9. Model Encapsulation

```
train_model <- function(X, y, num_classes, learning_rate, epochs, batch_size) {
  num_features <- ncol(X)

  # Initialize weights and biases
  weights <- matrix(runif(num_features * num_classes) - 0.5, nrow = num_features, ncol = num_classes)
  biases <- runif(num_classes) - 0.5

  for (epoch in 1:epochs) {
    # Mini-batch gradient descent
    for (i in seq(1, nrow(X), by=batch_size)) {
      batch_indices <- i:min(i+batch_size-1, nrow(X))
      X_batch <- X[batch_indices, ]
      y_batch <- y[batch_indices]

      # Forward propagation
      forward_result <- forward_propagation(X_batch, weights, biases)
      probs <- forward_result$probs

      # Compute loss (optional here, mainly for monitoring)
      loss <- cross_entropy_loss(probs, y_batch)
```

```
      # Compute gradients
      gradients <- compute_gradients(X_batch, y_batch, probs, num_classes)

      # Update parameters
      update_result <- update_parameters(weights, biases, gradients$dW, gradients$dB, learning_rate)
      weights <- update_result$weights
      biases <- update_result$biases
    }

    # Optionally evaluate the model on the training set or a validation set
    if (epoch %% 5 == 0) {
      forward_result <- forward_propagation(X, weights, biases)
      accuracy <- evaluate_accuracy(forward_result$probs, y)
      cat("Epoch", epoch, ": Training accuracy =", accuracy, "\n")
    }
  }

  return(list(weights=weights, biases=biases))
}
```

Explanation:

Initialization: We start by initializing the weights and biases with small random values. Training Loop: The outer loop iterates over the number of epochs. Each epoch represents a full pass through the training dataset. Mini-Batch Gradient Descent: Within each epoch, the dataset is divided into mini-batches. This approach helps to stabilize the gradient computation and can lead to faster convergence. Forward Propagation: For each batch, we compute the class probabilities using the current weights and biases. Loss Computation: While not strictly necessary for the update step, calculating the loss allows us to monitor training progress. Gradient Computation: We compute the gradients of the loss with respect to the weights and biases. Parameter Update: Using these gradients, we update the model parameters. Evaluation: Periodically, we evaluate the model's performance on the entire dataset (or a separate validation set) to monitor its accuracy. This function represents a basic framework for training a softmax regression model. In practice, you might enhance this process with additional features like validation checks, early stopping, learning rate schedules, or regularization to improve the model's performance and prevent overfitting.

10. Model Training

```
# Set hyperparameters
learning_rate <- 0.01
epochs <- 200  # Number of passes through the entire dataset
batch_size <- 100  # Number of training examples used in one iteration
num_classes <- length(unique(y))

# Train the model
model <- train_model(X, y, num_classes, learning_rate, epochs, batch_size)
```

```
## Epoch 5 : Training accuracy = 0.74235
## Epoch 10 : Training accuracy = 0.81295
## Epoch 15 : Training accuracy = 0.8384
## Epoch 20 : Training accuracy = 0.853
## Epoch 25 : Training accuracy = 0.86265
## Epoch 30 : Training accuracy = 0.8698
## Epoch 35 : Training accuracy = 0.87635
```

```
## Epoch 40 : Training accuracy = 0.88085
## Epoch 45 : Training accuracy = 0.8842
## Epoch 50 : Training accuracy = 0.88715
## Epoch 55 : Training accuracy = 0.8901
## Epoch 60 : Training accuracy = 0.8918
## Epoch 65 : Training accuracy = 0.89325
## Epoch 70 : Training accuracy = 0.89515
## Epoch 75 : Training accuracy = 0.89695
## Epoch 80 : Training accuracy = 0.8985
## Epoch 85 : Training accuracy = 0.8996
## Epoch 90 : Training accuracy = 0.9011
## Epoch 95 : Training accuracy = 0.90235
## Epoch 100 : Training accuracy = 0.90325
## Epoch 105 : Training accuracy = 0.9042
## Epoch 110 : Training accuracy = 0.90465
## Epoch 115 : Training accuracy = 0.9054
## Epoch 120 : Training accuracy = 0.9063
## Epoch 125 : Training accuracy = 0.90725
## Epoch 130 : Training accuracy = 0.90785
## Epoch 135 : Training accuracy = 0.9089
## Epoch 140 : Training accuracy = 0.90955
## Epoch 145 : Training accuracy = 0.91005
## Epoch 150 : Training accuracy = 0.9106
## Epoch 155 : Training accuracy = 0.91135
## Epoch 160 : Training accuracy = 0.912
## Epoch 165 : Training accuracy = 0.91275
## Epoch 170 : Training accuracy = 0.91335
## Epoch 175 : Training accuracy = 0.9141
## Epoch 180 : Training accuracy = 0.9148
## Epoch 185 : Training accuracy = 0.9155
## Epoch 190 : Training accuracy = 0.916
## Epoch 195 : Training accuracy = 0.9166
## Epoch 200 : Training accuracy = 0.917
```

11. Testing—data Preprocessing

```r
# Load the data
mnist_test <- read.csv("https://pjreddie.com/media/files/mnist_train.csv", skip=20000, nrows = 20000, h
colnames(mnist_test) <- c("Digit", paste("Pixel", seq(1:784), sep = ""))

# Normalize the pixel values to range [0, 1]
mnist_test[, -1] <- mnist_test[, -1] / 255

# Split the dataset into features (X) and labels (y)
X_test <- as.matrix(mnist_test[, -1])
y_test <- mnist_test$Digit
```

12. Model Evaluation

```r
# Perform forward propagation on the test set to get probabilities
forward_result_test <- forward_propagation(X_test, model$weights, model$biases)
probs_test <- forward_result_test$probs
```

```r
# Make predictions by selecting the class with the highest probability for each test example
predictions_test <- max.col(probs_test) - 1  # Adjust for R's 1-indexing

# Calculate the accuracy on the test set
accuracy_test <- sum(predictions_test == y_test) / length(y_test)

# Print the test accuracy
cat("Test Accuracy:", accuracy_test, "\n")
```

```
## Test Accuracy: 0.8988
```