SIMPLE NEURAL NETWORK

-Vignesh Pitchaiah-

Introduction

The purpose of this model is to train a neural network for classification tasks using the rectified linear unit (ReLU) activation function and the softmax function for output probabilities. This model includes functions for initializing the network parameters, forward and backward propagation, updating the parameters, computing the loss function, and predicting new outputs.

DATA-PREPOSSESSING

Basically we have prepared our data set of images for a model to use for training and testing. It is done by preprocessing each image to make it easier for the machine learning model to work with.
First, it defines a function called "preprocess image" that takes an image file path and image size as input, reads the image file, resizes it to the specified size, flattens it into a one-dimensional array, and converts it to a list of integers. Next, it sets the file paths for the training and validation image folders, and gets a list of the class labels from the training folder. It also initializes empty lists to store the preprocessed image data for the training and validation sets. Then, it loops over the images in the training folder, preprocesses each image using the "preprocess image" function, adds the class label to the beginning of the preprocessed image data, and appends the result to the training data list. It does the same for the validation set. After all the images have been preprocessed and added to the training and validation data lists, it writes the data to CSV files with a header row that specifies the label and pixel values for each image.
Finally, it loads the data from the CSV files into NumPy arrays for use in a model.
Then we read the data from the CSV file
The data is then shuffled randomly, to help prevent over-fitting during training. The data is then split into training and validation sets, with the first 200 samples being used for validation, and the rest being used for training. The labels for the validation and training data are separated from the pixel values and stored in separate variables.
Finally, the pixel values are normalized by dividing by 255 (since pixel values range from 0 to 255), and then converted to integers. By doing this we are making it ready to be used for training our model.

MODEL ARCHITECTURE

In this built neural network from scratch we implemented its training and prediction functions. First we initializes the weights and biases of a deep neural network with random values. The parameters are represented by matrices W and biases b.

- init-params(): The function initializes the parameters of the neural network with random values. For each layer i, it creates a weight matrix Wi of size (ni, ni+1) and a bias vector bi of size (1, ni+1), where ni and ni+1 are the number of nodes in layer i and i+1 respectively. The values are generated randomly using a uniform distribution between -0.5 and 0.5, and then shifted so that they are centered around zero by subtracting 0.5 from the generated values.
  Formula: W-i = np.random.uniform(low=-0.5, high=0.5, size=(n-i, n-i+1)) - 0.5
  b-i = np.random.uniform(low=-0.5, high=0.5, size=(1, n-i+1)) - 0.5

- ReLU(Z): The rectified linear unit (ReLU) activation function takes a matrix Z as input and returns a matrix of the same dimensions. The function replaces any negative values in the input matrix with zero, and leaves positive values unchanged. This function is used in the network to introduce non-linearity.
  Formula: ReLU(Z) = max(0, Z)

- softmax(Z): The softmax function takes a matrix Z as input and returns a matrix of the same dimensions. It is used in the output layer of the network to convert the output values into a probability distribution. The function first computes the exponential of each element in the input matrix, and then normalizes the results so that they sum to one.

Formula: softmax(Z) = exp(Z) / sum(exp(Z))

- forward-prop(): The function takes in the initialized parameters of the network (W1, b1, W2, b2, etc.) as well as an input data matrix X. It then computes the forward propagation through the network, starting with the first layer and ending with the final layer. It returns the intermediate values (Z1, A1, Z2, A2, etc.) as well as the final output of the network (A5).
  Formula: Z-i = A-(i-1) * W-i + b-i
  A-i = ReLU(Z-i) for all layers except the last
  A-i = softmax(Z-i) for the last layer
  A5 = final output of the network

- ReLU-deriv(Z): This is the derivative of the rectified linear unit activation function (ReLU) that is used in the backward propagation step of the network. It takes a matrix Z as input and returns a matrix of the same dimensions, where any negative values in Z are replaced with zeros.
  Formula: ReLU-deriv(Z) = 1 if Z ¿= 0, and 0 otherwise

- one-hot(Y): This function takes a vector of labels Y and converts it into a one-hot encoding format. For each label in Y, the function creates a row vector of length k (the number of classes), where all values are zero except for the value at the index corresponding to the label, which is set to one.
  Formula: one-hot(Y) = matrix of size (len(Y), k), where
  each row is a row vector of length k with a one at the index corresponding to the label in Y

- backward-prop(): This function computes the gradients of the loss function with respect to the parameters of the network. It takes in the intermediate values (Z1, A1, Z2, A2, etc.), the initialized parameters of the network (W1, b1, W2, b2, etc.), the input data matrix X, and the labels Y. It uses the chain rule of differentiation to compute the gradients of the loss function with respect to each parameter. The gradients are then used to update the parameters in the opposite direction of the gradient, in order to minimize the loss function. The function first computes the derivative of the loss function with respect to the output of the network (dA5), using the cross-entropy loss function. It then uses this value to compute the derivatives of the output layer (dZ5, dW5, db5), and propagates these derivatives backwards through the network to compute the derivatives of the previous layers (dZ4, dW4, db4, etc.).
  Formula: dA5 = A5 - one-hot(Y)
  dZ5 = dA5
  dW5 = A4.T * dZ5
  db5 = sum(dZ5, axis=0, keepdims=True)
  dA4 = dZ5 * W5.T
  dZ4 = dA4 * ReLU-deriv(Z4)
  dW4 = A3.T * dZ4
  db4 = sum(dZ4, axis=0, keepdims=True)
  dA3 = dZ4 * W4.T
  dZ3 = dA3 * ReLU-deriv(Z3)
  dW3 = A2.T * dZ3
  db3 = sum(dZ3, axis=0, keepdims=True)
  dA2 = dZ3 * W3.T
  dZ2 = dA2 * ReLU-deriv(Z2)
  dW2 = A1.T * dZ2
  db2 = sum(dZ2, axis=0, keepdims=True)
  dA1 = dZ2 * W2.T
  dZ1 = dA1 * ReLU-deriv(Z1)
  dW1 = X.T * dZ1
  db1 = sum(dZ1, axis=0, keepdims=True)

- update-params(): This function takes in the initialized parameters of the network (W1, b1, W2, b2, etc.) as well as the computed gradients of the loss function with respect to each parameter (dW1, db1, dW2, db2, etc.). It then updates the parameters using the gradients and a learning rate (alpha).
  Formula: W-i = W-i - alpha * dW-i
  b-i = b-i - alpha * db-i

Here, alpha is the learning rate, which determines the size of the steps taken in the opposite direction of the gradients during parameter updates. The larger the learning rate, the larger the steps taken, and the faster the network converges. However, if the learning rate is too large, the network may overshoot the minimum of the loss function and diverge. The update-params() function updates each parameter of the network using the gradients and the learning rate. This process is repeated for a fixed number of iterations or until the loss function reaches a certain threshold. After training, the network can be used to make predictions on new data by forward propagating the input through the network and returning the output of the last layer.

- compute-loss: This function takes in the predicted outputs of the network and the true labels, and computes the cross-entropy loss between them.
  Formula: L = -1/m * sum(Y * log(A5) + (1-Y) * log(1-A5))
  where: L is the cross-entropy loss, m is the number of training examples, Y is the true label vector (one-hot encoded), A5 is the predicted output of the network

- predict: This function takes in a set of inputs, and uses the trained parameters of the network to predict the outputs.
  Formula: Z1 = X * W1 + b1
  A1 = ReLU(Z1)
  Z2 = A1 * W2 + b2
  A2 = ReLU(Z2)
  Z3 = A2 * W3 + b3
  A3 = ReLU(Z3)
  Z4 = A3 * W4 + b4
  A4 = ReLU(Z4)
  Z5 = A4 * W5 + b5
  A5 = sigmoid(Z5)
  predictions = (A5 ¿ 0.5)
  where: X is the input matrix, W1, b1, W2, b2, W3, b3, W4, b4, W5, b5 are the trained parameters of the network ReLU and sigmoid are activation functions predictions is a vector of binary values (0 or 1), indicating the predicted class for each input example

- train: This function performs the forward and backward propagation steps, and updates the parameters of the network using gradient descent.
  Formula: here num-iterations is the number of iterations to run the optimization algorithm,X is the input matrix, Y is the true label vector (one-hot encoded),
  parameters is a dictionary containing the initialized parameters of the network cache is a dictionary containing the intermediate values computed during forward propagation grads is a dictionary containing the gradients of the loss function with respect to each parameter learning-rate is the step size used in the update rule for gradient descent print-cost is a flag indicating whether to print the cost after every 1000 iterations

TRAINING AND USE

During training, the network is initialized using the init-params() function, and the train() function is called with the input data matrix X, the labels Y, and a number of training epochs. The train() function updates the parameters of the network using the gradients computed in the backward=prop() function and the learning rate specified by the user. The performance of the model during training can be monitored by computing the loss function using the compute-loss() function.
After training, the trained parameters of the network can be used to predict new outputs using the predict() function. This function takes in an input data matrix X and the trained parameters of the network, computes the forward propagation through the network, and returns the predicted labels for each input in X

PERFORMANCE

The effectiveness of the model can be evaluated using metrics such as accuracy or F1 score. Our model was trained with a learning rate of 0.07, and after 900 epochs, we achieved an accuracy of 99.875 percent on the training set. During prediction, the model correctly identified the label as 5. Furthermore, we validated the model on a test set and achieved an accuracy of 98.50 percent, a precision score of 0.99, and an F1 score of 0.98. The confusion matrix for the validation set shows that the model performed well on most classes, with a few misclassifications in a few others. We also tested the model on unseen data, where it

achieved a precision score of 0.8209339627259264, a recall score of 0.8017320261437908, and an F1 score of 0.8015037501705734. The confusion matrix for the unseen data shows that the model performed well overall, with only a few misclassifications in some classes.

Metric	Training Set	Validation Set	Unseen Data
Accuracy	99.875%	98.50%	80.17%
Precision	0.90	0.99	0.82
Recall	0.90	0.91	0.80
F1 Score	0.95	0.98	0.80

Table 1: Model Performance Metrics

CONCLUSION

Based on the performance of our model on both the training and testing data, it can be concluded that the model is performing quit well on trained and validation process than un seen data. The model achieved a high accuracy of 99.875 percentage on the training data and an accuracy of 85.5 percentage on the validation data. The precision score and F1 score were also high, indicating that the model's predictions were reliable.When tested on unseen data, the model achieved a precision score of 82.1 percentage, recall score of 80.2 percentage and an F1 score of 80.2 percentage, which suggests that the model is able to generalize well on new data. The confusion matrix shows that the model made few errors in prediction.

Overall, the model appears to be well-trained and capable of accurately predicting the target variable. However, I consider that further testing on new data and refinement of the model may be necessary for optimal performance because evaluation on training is high than testing.