Jacob Freedman

DS210 Project Report

Overview:

The goal of this project is to answer the question of if you can infer what activity someone is performing based on smartwatch data. The dataset I used is from kaggle and is too large to be put on Github. It contains data from inertial sensors in smartwatches from a 2013 study comprised of 30 participants.

Data Processing:

After I had downloaded the dataset from kaggle, I moved the files from my desktop to my project folder. In my main file, I created a module to read the csv information into my code. The data set was initially very clean, and after some exploratory data analysis I found there were no missing values that I had to fill. The only preprocessing I did was encoding the 6 different activity options so that my model could treat them as quantitative data.

Modules:

For this project, I decided to segment my code between one main file and two modules. The first module "data_loader.rs" is what I used to store all the code needed for reading my CSV into data type compatible for rust operations. I figured this function was best in a separate module as it was not exclusively related to the neural network's implementation. The second module I made was "nn_tester.rs". This was where I stored all the tests I used to check my final model. I found it beneficial to separate the tests from the main file, as it left the main file only with the neural network's code which decluttered the file's program environment greatly.

Key Functions & Types:

The first thing in the code I created was a struct "NeuralNetwork" that I used to store the internal parameters of the neural networks. This struct became crucial for managing the weights between the forward pass and backwards propagation. Next, I implemented a "new" method for this struct that I used to initialize a new neural network. It takes in the inputs of input size, hidden1_size and hidden2_size, as well as the output size and learning rate. The main body works by using an array of random numbers between -0.5 and 0.5 to initialize the weights.

Next, I implemented a simple sigmoid function to serve as the activation function, as well as its derivative to support gradient computation during backpropagation. These functions are applied to the weighted sums and return activations or gradients, which are essential for adjusting weights in the backward pass.

I implemented the softmax function because we are dealing with a multi-class classification problem. Its purpose is to convert the raw output scores from the final layer into probabilities. The function takes a 2D array of logits and transforms each row into a normalized probability distribution, where all values in a row sum to 1.

The forward pass computes activations for each layer using matrix multiplication and activation functions. It takes the input, applies weighted sums and the sigmoid function for hidden layers, then applies softmax to the output layer, returning activations for each layer.

The backward pass computes gradients using the loss between predictions and true labels. It applies the chain rule to propagate errors backward through the network, updating weights based on the gradients computed from each layer's activations and the learning rate.

All these functions marked the main implementation. From here on out, I created functions to help me evaluate and interpret my results. The first I made was a predict function which ran a forward pass and selected the index of the highest probability output. For this it took in the input data and returned the maximum indices. The accuracy function compares the predicted class labels to the true class labels in the dataset. It first uses the predict function to classify the input data, obtaining the class with the highest predicted probability. Then, it compares these predicted labels with the actual labels and calculates the proportion of correct predictions. This ratio is the model's accuracy on the test set or validation set. Finally, I made a function to see the accuracy in each individual class and I did this by taking in all the true and predicted labels, as well as the classes. I then initialize a vector to store the number of correctly predicted and number of samples for each class. I iterated over each prediction and corresponding target and then divided these two vector sums and printed the results for each class.

Tests:

The first test checks if the load_data function works correctly by creating dummy data, ensuring that the feature matrix has the correct dimensions and the labels are parsed accurately. The second test verifies the correctness of the sigmoid function and its derivative; it checks that the sigmoid at position (0,0) outputs

0.5 (as expected for input 0), and that the derivative outputs 0.25 at the same position, which is the maximum value of the sigmoid's derivative. Finally, the third test validates the forward pass by checking the shapes of the activation outputs for a given input, ensuring that the dimensions match the expected shapes for each layer of the network.

Output:

running 3 tests

test nn_tester::test_sigmoid_and_derivative ... ok

test nn_tester::test_forward_shape ... ok

test nn_tester::test_load_data ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in

Results:

Epoch 3/3

Overall Accuracy: 90.16%

Class Accuracy:

WALKING → 482 / 496 = 97.18%

WALKING_UPSTAIRS → 416 / 471 = 88.32%

WALKING_DOWNSTAIRS → 392 / 420 = 93.33%

SITTING → 355 / 491 = 72.30%

STANDING → 475 / 532 = 89.29%

LAYING → 537 / 537 = 100.00%

The results from running cargo run --release show high accuracy, indicating that the inertial sensor data from the smartwatches is highly predictive of the activity being performed. Notably, for the "laying" class,

we achieved a perfect accuracy of 100%, which raised concerns about potential overfitting. However, after experimenting with different epochs and learning rates, the accuracy for "laying" consistently remained at 100%. I concluded that this outcome is likely due to the nature of the "laying" activity, which involves minimal movement, making it easier for the model to distinguish from the other activities. The lower movement in this class results in more distinct sensor patterns, enabling the model to differentiate it effectively, while the other activities, with more variable movement, are harder to classify with such certainty.

Usage instructions:

For this program, I would recommend using “cargo run --release” to speed up run time as otherwise it will take a while to return output. With the optimized release however, the program should only take a few seconds.

Citations:

For this project I relied heavily on homework #9 and Lecture 22 with professor Thomas Gardos to help me gain understanding of Neural Networks and their implementation in Rust.