CSE 625 Term Project Report

Digits Prediction of MNIST Dataset using Multithreading

Youssef Sheta

07-29 -2023

Objective

The goal of this project is to utilize multithreading of Julia language and use it in training models in MNIST dataset. I am using Flux.jl library to parallelize training process of CNN (Convolutional Neural Network) for digit classification. Moreover, I am comparing the difference between sequentially training a model with using multithreaded model to evaluate the efficiency of using multithreading.

General Description of the Approach

This project using Julia programming language, utilizing its capabilities in multithreading, and using it in training process of a deep learning model, which is detecting what digit is written in an image in MNIST dataset. Julia language is a great choice for this project due to its high-performance and scientific computing tasks beside providing distributed parallel execution, and an extensive mathematical function library.

Platforms

Flux.jl: A machine learning library.

MLDatasets.jl: A package that offers common machine learning datasets like MNIST dataset, which is used in this project.

Base.Threads.jl: Library to use threads in training my model.

Implementation Details

Data Loading and Preprocessing

Firstly, MNIST dataset is loaded and preprocessed. Images and labels are loaded using MNIST.traindata function. An extra dimension is added to the images and labels, which are values from 0 to 9, are converted into one hot format. Then I divided the data in to batches using ‘DataLoader’ function, I divided the data into mini batches because it is an efficient way when training neural networks because it speeds up training and also to optimize the memory usage as large datasets may

not fit into the computer's memory, so you will have to load a small portion of the data into the memory at a time.

```
using Flux
using Flux.Data: DataLoader
using Flux: onecold, onehotbatch, crossentropy, throttle
using MLDatasets: MNIST
using Statistics

# Load MNIST dataset
imgs, labels = MNIST.traindata(Float32);

# Preprocess images and labels
imgs = Flux.unsqueeze(imgs, 3);
labels = onehotbatch(labels, 0:9);

# Partition into batches of size 1,000
data = DataLoader((imgs, labels), batchsize=1000, shuffle=true);
```

Model Definition

```
# Define the model
model = Chain(
    Conv((2,2), 1=>16, relu),
  MaxPool((2,2)),
  Conv((2,2), 16=>8, relu),
  MaxPool((2,2)),
  x -> reshape(x, :, size(x, 4)),
  Dense(288, 10),
  softmax)
```

The model has two convolutional layers with ReLU activation functions and two max-pooling layers. The output for the last max-pooling layer is reshaped to 2d tensor that is passed to a dense layer. The softmax activation function is used to output the probability of each class, in which it outputs the probabilities of each digit.

Training the Model Sequentially

```
# Loss function and evaluation metric
loss(x, y) = crossentropy(model(x), y)
accuracy(x, y) = mean(onecold(model(x)) .== onecold(y))

# Optimization parameters
opt = ADAM(0.01)
```

```
# Train the model for 15 epochs
elapsed_time = @elapsed begin
for epoch = 1:15
  Flux.train!(loss, Flux.params(model), data, opt)
  @show accuracy(tX, tY)
end
end

println("Test accuracy: ", accuracy(tX, tY))
println("Elapsed time: $elapsed_time seconds")
```

The train! function is used to train the model.

Training the model with Multithreading

In [109]:
```
# Train the model for 15 epochs
# Convert data to array format
data_array = [(x, y) for (x, y) in data]

function split_data(data, chunks)
    len = length(data)
    chunk_size = len ÷ chunks
    return [data[i:min(i+chunk_size-1, len)] for i in 1:chunk_size:len]
end

elapsed_time = @elapsed begin
 for epoch = 1:15
    # Split data into chunks that will be trained separately
    data_chunks = split_data(data_array, nthreads())

    # Array to hold the tasks
    tasks = Task[]

    # Spawn a new task for each data chunk
    for chunk in data_chunks
        push!(tasks, @spawn Flux.train!(loss, Flux.params(model), chunk, opt))
    end

    # Wait for all tasks to complete
    foreach(wait, tasks)

    @show accuracy(tX, tY)
 end
end

println("Test accuracy: ", accuracy(tX, tY))
println("Elapsed time: $elapsed_time seconds")
```

Here the data is split into chunks and then separate threads train each chunk. Chunks are trained separately, and their number is equal to the available number of threads, which is 4 in this project.

In [67]:
```
using Base.Threads
nthreads()
```

Out[67]: 4

Time performance analysis

The total elapsed time for sequential approach is 109.65 seconds and using threads is 60 seconds.

The time saved by using threads is approximately 49.5 seconds, which means that multithreading was able to reduce the training time and optimize machine learning workflows. The speedup is approximately 1.82 faster than the sequential approach. This project managed to achieve the expected speedup using threads although it was expected to be 4 times faster, but in practice, due to overheads, the ideal speedup is not often possible.

Result Presentation

As mentioned before the goal for this project is to compare two different methods for training a machine learning model to predict handwritten digits from MNIST dataset. When testing the trained model with actual images of MNIST dataset, I get the same result which is 8, but faster in the multithreading approach.

```
In [107]: using Images, ImageTransformations
          # Load the image
          image = load("/Users/youssef/Downloads/archive (3)/testSet/testSet/img_3.jpg")

          # Convert the image to grayscale
          image = Gray.(image)

          # Resize the image to the dimensions expected by the model (28x28)
          image = imresize(image, (28, 28))

          # Convert the image to float32
          image = Float32.(image)

          # Reshape the image into the 4D tensor format (height x width x color_channels x num_images)
          image = reshape(image, 28, 28, 1, 1)
```

This code shows how I load the image from the downloaded dataset and preprocess it to pass it to the model to make a prediction for the handwritten image to a certain number between 0 and 9.

```
In [108]: prediction = model(image)
          predicted_class = argmax(prediction)[1] - 1  # Subtract 1 because classes start from 0

Out[108]: 8
```

This is the output for the sequential approach and give us a good accuracy.

```
accuracy(tX, tY) = 0.95
accuracy(tX, tY) = 0.9517
accuracy(tX, tY) = 0.9517
accuracy(tX, tY) = 0.9543
accuracy(tX, tY) = 0.9553
accuracy(tX, tY) = 0.9541
accuracy(tX, tY) = 0.9548
accuracy(tX, tY) = 0.9565
accuracy(tX, tY) = 0.9564
accuracy(tX, tY) = 0.9574
accuracy(tX, tY) = 0.9584
accuracy(tX, tY) = 0.9587
accuracy(tX, tY) = 0.9591
accuracy(tX, tY) = 0.9593
accuracy(tX, tY) = 0.96
Test accuracy: 0.96
Elapsed time: 109.647311833 seconds
```

While this output was using multithreading approach.

```
accuracy(tX, tY) = 0.9594
accuracy(tX, tY) = 0.9164
accuracy(tX, tY) = 0.9571
accuracy(tX, tY) = 0.9555
accuracy(tX, tY) = 0.9587
accuracy(tX, tY) = 0.9564
accuracy(tX, tY) = 0.9544
accuracy(tX, tY) = 0.9557
accuracy(tX, tY) = 0.9552
accuracy(tX, tY) = 0.9589
accuracy(tX, tY) = 0.9555
accuracy(tX, tY) = 0.956
accuracy(tX, tY) = 0.9555
accuracy(tX, tY) = 0.9358
accuracy(tX, tY) = 0.9573
Test accuracy: 0.9573
Elapsed time: 60.148196125 seconds
```

They both have approximately the same accuracy and they also have the same output in most of the images, the only difference is that the multithreading approach shows how threads can be useful in speeding up the process when dealing with large datasets and can lead to substantial performance improvements, so it reduce computation time while maintaining the same level of prediction accuracy.

Contributions

I preprocessed the MNIST dataset and applied one hot encoding to the labels to get the data in correct form. Then I constructed a CNN using Flux package in Julia. I performed model training using two approaches as stated previously in the results part. Moreover, I applied multithreading to speed up the training process of the model. Lastly, I validated the model's performance using a test dataset from the MNIST dataset and ensured that both models gave the same result with better execution time.

References

Svaksha. "SVAKSHA/Julia.Jl: Curated Decibans of Julia Programming Language." *GitHub*, github.com/svaksha/Julia.jl. Accessed 27 July 2023.