Julia-NN from Scratch

November 3, 2022

1 NN from Scratch

This project came to be as the final assignment in a course I took in university. At the time the plan was to port the code from Python to R, which was a cumbersome challenge in itself. Since then I've been trying to learn Julia and what better way to learn than trying to port something I am very familiar with at this point? I will keep the same introduction as I used in the other two notebooks in case anyone ever lands on this notebook first.

As a conclusion to the STAN48 course I decided to create simple implementation of a feed forward neural network using mainly Numpy and base Python. While there are already exquisite packages that offer these solutions (like Tensorflow and Pytorch), a step by step implementation of a neural network is still valuable for teaching basic programming concepts as well as basic neural network concepts. The code and explanations presented here are inspired heavily from two sources, namely Andrew Ng's course on Neural Networks and Deep Learning, and LUSEM's Deep Learning and AI Methods course. The code is then an adaptation of the teachings found in both courses. Additionally, in this project I offer an R version for the project presented here, which can be found in the `R-NN_from_scratch.ipynb` file. As a final disclaimer I must admit that adapting the code in Python was not a hard task, but porting it to R was a strenuous nightmare-like task since the data types can be treated quite differently in both R and Python.

As a general example I will use the Kaggle Dogs vs. Cats dataset to classify whether a given picture shows a cat or not. As the data set only includes two different options, we can assume the `not cat` option to be the same as `dog`. As mentioned, the intent is to have a general example to expose how the algorithm works, and the intricasies of the programming challenge, in other words, it is *not* my intention to implement a functioning neural network from scratch **and** a good model for classifying cats.

1.1 R or Python? ... Or Julia?

This notebook is originally made for Python. One of the requirements for this project was that whatever the choice of application to be developed, it should be done in both Python **and** R. Well, that I did, now I'm doing it for Julia too cause why not. You can check the notebooks for Python an R in this same repository.

1.1.1 The structure

This project includes several files. In this notebook you will find the application related functions, however, many of the base functions used for the calculations are left in a separate file that concentrates all the basic calculation functions. Without those dependencies this notebook will not function as it should. Some basic concepts regarding neural networks will be presented through

the notebook, but the focus of this work is exposing the programming challenge behind neural networks.

1.1.2 The data

The data set contains 25000 images of dogs and cats, but 59 of them were corrupted or in grayscale and, therefore, dropped. The classes are balanced and the angle, depth, light, and dimensions are not uniform. While originally a Kaggle competition data set, I opted to use the version made available by Microsoft because it did not pre divide the data giving me more freedom to split the sets as I please.

[1]:
```
function relu(Z)
  A = max.(Z, 0)
  cache = Z
  @assert size(A) == size(Z) "Sizes don't match in relu function"
  return A, cache
end

function sigmoid(Z)
  A = 1 ./ (1 .+ exp.(Z))
  cache = Z
  return A, cache
end

function relu_backprop(dA, cache)
  Z = cache
  dZ = dA
  dZ[Z .<= 0] .= 0
  @assert size(dZ) == size(Z) "Sizes don't match in relu_backprop function"
  return dZ
end

function sigmoid_backprop(dA, cache)
  Z = cache
  s = 1 ./ (1 .+ exp.(-Z))
  dZ = dA .* s .* (1 .- s)
  @assert size(dZ) == size(Z) "Sizes don't match in sigmoid_backprop function"
  return dZ
end
```

[1]:
```
sigmoid_backprop (generic function with 1 method)
```

[2]:
```
write("helper_functions.jl", In[IJulia.n - 1])
```

[2]:
```
581
```

[3]:
```
function init_param(layer_dim)
  L = length(layer_dim)
```

```
  parameters = Dict()
  for l in 2:(L)
    parameters[string("W", l-1)] = rand(layer_dim[l], layer_dim[l - 1]) * 0.01
    parameters[string("b", l-1)] = zeros(layer_dim[l], 1)
    @assert size(parameters[string("W", l-1)]) == (layer_dim[l], layer_dim[l -␣
 ↪1]) "Weights size wrong in init_param function"
    @assert size(parameters[string("b", l-1)]) == (layer_dim[l], 1) "Bias size␣
 ↪wrong in init_param function"
  end
  return parameters
end
```

[3]: init_param (generic function with 1 method)

[4]:
```
write("init_param.jl", In[IJulia.n - 1])
```

[4]: 496

[5]:
```
include("helper_functions.jl")

function for_prop(A, W, b)
  Z = (W * A) .+ b
  @assert size(Z) == (size(W)[1], size(A)[2])
  cache = (A, W, b)
  return Z, cache
end

function for_activation(A_prev, W, b, activ)
  if activ == "sigmoid"
    Z, linear_cache = for_prop(A_prev, W, b)
    A, activ_cache = sigmoid(Z)
  elseif activ == "relu"
    Z, linear_cache = for_prop(A_prev, W, b)
    A, activ_cache = relu(Z)
  end
  @assert size(A) == (size(W)[1], size(A_prev)[2]) "Activation size wrong in␣
 ↪for_activation function"
  cache = (linear_cache, activ_cache)
  return A, cache
end
```

[5]: for_activation (generic function with 1 method)

[6]:
```
write("for_prop.jl", In[IJulia.n - 1])
```

[6]: 579

[7]:
```
include("for_prop.jl")

function deep_model(X, parameters)
  caches = []
  A = X
  L = div(length(parameters), 2)
  for l in 1:(L-1)
    A_prev = A
    A, cache = for_activation(A_prev,
                              parameters[string("W", l)],
                              parameters[string("b", l)],
                              "relu")
    push!(caches, cache)
  end
  AV, cache = for_activation(A,
                             parameters[string("W", L)],
                             parameters[string("b", L)],
                             "sigmoid")
  push!(caches, cache)
  @assert size(AV) == (1, size(X)[2]) "AV size wrong in deep_model function"
  return AV, caches
end
```

[7]:
```
deep_model (generic function with 1 method)
```

[8]:
```
write("deep_model.jl", In[IJulia.n - 1])
```

[8]:
```
680
```

[9]:
```
function cost_computation(AV, Y)
  Y = Y'
  m = size(Y)[2]
  cost = -(1/m) * sum(log.(AV) * Y' .+ log.((1 .- AV)) * (1 .- Y'))
  return cost
end
```

[9]:
```
cost_computation (generic function with 1 method)
```

[10]:
```
write("cost_computation.jl", In[IJulia.n - 1])
```

[10]:
```
144
```

[11]:
```
include("helper_functions.jl")

function back_prop(dZ, cache)
  A_prev, W, b = cache
  m = size(A_prev)[2]
  dW = (1/m) * dZ * A_prev'
```

```
  db = (1/m) * sum(dZ, dims=2)
  dA_prev = W' * dZ
  @assert size(dA_prev) == size(A_prev) "dA_prev size wrong in back_prop␣
 ↪function"
  @assert size(dW) == size(W) "dW size wrong in back_prop function"
  @assert size(db) == size(b) "db size wrong in back_prop function"
  return dA_prev, dW, db
end

function back_activ(dA, cache, activ)
  for_cache, activ_cache = cache
  if activ == "relu"
    dZ = relu_backprop(dA, activ_cache)
    dA_prev, dW, db = back_prop(dZ, for_cache)
  elseif activ == "sigmoid"
    dZ = sigmoid_backprop(dA, activ_cache)
    dA_prev, dW, db = back_prop(dZ, for_cache)
  end
  return dA_prev, dW, db
end

function deep_model_back(AV, Y, caches)
  grads = Dict()
  L = length(caches)
  m = size(AV)[2]
  Y = reshape(Y, 1, length(AV))
  dAL = -(Y ./ AV) - ((1 .- Y) ./ (1 .- AV))
  present_cache = caches[L]
  grads[string("dA", L-1)], grads[string("dW", L)], grads[string("db", L)] =␣
 ↪back_activ(dAL, present_cache, "sigmoid")
  for l in (L - 2):-1:0
    present_cache = caches[l+1]
    dA_prev_temp, dW_temp, db_temp = back_activ(grads[string("dA", l+1)],␣
 ↪present_cache, "relu")
    grads[string("dA", l)] = dA_prev_temp
    grads[string("dW", l+1)] = dW_temp
    grads[string("db", l+1)] = db_temp
  end
  return grads
end
```

[11]:
```
deep_model_back (generic function with 1 method)
```

[12]:
```
write("back_prop.jl", In[IJulia.n - 1])
```

[12]:
```
1385
```

[13]:
```
function update(params, grads, learning_rate)
  parameters = copy(params)
  L = div(length(parameters), 2)
  for l in 1:L
    parameters[string("W", l)] = parameters[string("W", l)] - learning_rate *␣
 ↪grads[string("dW", l)]
    parameters[string("b", l)] = parameters[string("b", l)] - learning_rate *␣
 ↪grads[string("db", l)]
  end
  return parameters
end
```

[13]:
```
update (generic function with 1 method)
```

[14]:
```
write("update.jl", In[IJulia.n - 1])
```

[14]:
```
353
```

[15]:
```
using Images, FileIO, InvertedIndices, Suppressor

function process_image(path_vec::Vector{String}, h::Int64, w::Int64, label::
 ↪Int64)
  result = zeros((h*w*3), length(path_vec))
  class = Int[]
  @suppress begin
    for i in enumerate(path_vec)
      try
        img = load(i[2])
      catch
        continue
      end
      img = imresize(img,(h,w))
      try
        img = reshape(channelview(img), ((h*w*3), 1)) # [temp(img[i]) for i = 1:
 ↪length(img), temp in (red, green, blue)]
      catch
        continue
      end
      result[:,i[1]] = img
      push!(class, label)
    end
  end
  return result, class
end

function create_dataset(filenames_cat::Vector{String}, filenames_dog::
 ↪Vector{String}, height::Int64, width::Int64, labels)
```

```
  cat_i, cat_l = process_image(filenames_cat, height, width, labels[1])
  dog_i, dog_l = process_image(filenames_dog, height, width, labels[2])
  imgs = hcat(cat_i, dog_i)
  class = vcat(cat_l, dog_l)
  i=1
  while i <= size(imgs)[2]
    imgs = sum(imgs[:,i]) == 0 ? imgs[:, Not(i)] : imgs
    i +=1
  end
  return imgs, class
end
```

[15]: create_dataset (generic function with 1 method)

[16]:
```
write("create_dataset.jl", In[IJulia.n - 1])
```

[16]: 1071

[17]:
```
include("update.jl")
include("back_prop.jl")
include("deep_model.jl")
include("init_param.jl")
include("cost_computation.jl")
function dense_nn(X, Y, layers_dims, learning_rate, num_iterations, print_cost)
  costs = Float64[]
  parameters = init_param(layers_dims)
  for i in 1:num_iterations
    AV, caches = deep_model(X, parameters)
    cost = cost_computation(AV, Y)
    grads = deep_model_back(AV, Y, caches)
    parameters = update(parameters, grads, learning_rate)
    if print_cost && i%100==1 || i==num_iterations-1
      println(string("Cost after iteration ",i,": ",cost))
    end
    if i%100==1 || i==num_iterations
      push!(costs, cost)
    end
  end
  return parameters, costs
end
```

[17]: dense_nn (generic function with 1 method)

[18]:
```
write("dense_nn.jl", In[IJulia.n - 1])
```

[18]: 698

[19]:
```
include("deep_model.jl")
function predict(X, y, parameters)
  m = size(X)[2]
  n = div(length(parameters), 2)
  p = Int[]
  probs, caches = deep_model(X, parameters)
  for i in 1:size(probs)[2]
    if probs[i] > 0.25
      push!(p, 1)
    else
      push!(p, 0)
    end
  end
  print(string("Accuracy ", sum((p .== y)/m)))
  return p
end
```

[19]:
```
predict (generic function with 1 method)
```

[20]:
```
write("predict.jl", In[IJulia.n - 1])
```

[20]:
```
337
```

[21]:
```
include("create_dataset.jl")

cat_path = "C:/Users/wtrindad/source/repos/NN_from_scratch/PetImages/Cat/"
cat_imgs = joinpath.(cat_path, readdir(cat_path))
dog_path = "C:/Users/wtrindad/source/repos/NN_from_scratch/PetImages/Dog/"
dog_imgs = joinpath.(dog_path, readdir(dog_path))

img_data, img_label = create_dataset(cat_imgs, dog_imgs, 32, 32, (1,0));
```

```
Corrupt JPEG data: 239 extraneous bytes before marker 0xd9
Corrupt JPEG data: 214 extraneous bytes before marker 0xd9
Corrupt JPEG data: 128 extraneous bytes before marker 0xd9
Corrupt JPEG data: 99 extraneous bytes before marker 0xd9
Corrupt JPEG data: 1153 extraneous bytes before marker 0xd9
Corrupt JPEG data: 396 extraneous bytes before marker 0xd9
Corrupt JPEG data: 228 extraneous bytes before marker 0xd9
Corrupt JPEG data: 162 extraneous bytes before marker 0xd9
Warning: unknown JFIF revision number 0.00
Warning: unknown JFIF revision number 0.00
Corrupt JPEG data: 1403 extraneous bytes before marker 0xd9
Corrupt JPEG data: 252 extraneous bytes before marker 0xd9
Corrupt JPEG data: 2226 extraneous bytes before marker 0xd9
Corrupt JPEG data: 65 extraneous bytes before marker 0xd9
```

[22]:
```
using StatsBase
samples = wsample([1, 0], Weights([0.85, 0.15]), size(img_data)[2],␣
 ↪replace=true)

x_train = img_data[:, samples .== 1]
y_train = img_label[samples .== 1]
x_test = img_data[:, samples .== 0]
y_test = img_label[samples .== 0];
```

```
WARNING: using StatsBase.predict in module Main conflicts with an existing
identifier.
```

[23]:
```
@show size(x_train)
@show length(y_train)
@show size(x_test)
@show length(y_test);
```

```
size(x_train) = (3072, 21185)
length(y_train) = 21185
size(x_test) = (3072, 3780)
length(y_test) = 3780
```

[24]:
```
layer_dims = [size(x_train)[1],20,7,5,1]
```

[24]:
```
5-element Vector{Int64}:
 3072
   20
    7
    5
    1
```

[25]:
```
include("dense_nn.jl")
parameters, costs = dense_nn(x_train, y_train, layer_dims, 0.01, 1, true)
@show costs;
```

```
Cost after iteration 1: 0.6931448815914915
costs = [0.6931448815914915]
```

[26]:
```
parameters, costs = dense_nn(x_train, y_train, layer_dims, 0.01, 150, true);
```

```
Cost after iteration 1: 0.6931453423709729
Cost after iteration 101: 0.7253819413946185
Cost after iteration 149: 0.7688382433655048
```

[27]:
```
include("predict.jl")
predict_train = predict(x_train, y_train, parameters);
```

```
Accuracy 0.5008732593816377
```

[28]:
```
predict_test = predict(x_test, y_test, parameters);
```

```
Accuracy 0.49576719576719547
```

1.1.3 Note:

Porting everything to Julia has been quite fun and didactical to be honest, but some questions remain. While in Python and R I could use a threshold of 0.5 for the predictions, in Julia I had to settle for 0.25 because no probability would go above 0.3. This could be evidence that there is a bug, but could also be evidence of an exponential distribution of activation levels on the sigmoid, which in turn would cause the model to be extra sensitive around 0.20 - 0.30. As an exceptionally talented friend of mine explained to me recently, nowhere it is said that the probabilities should land evenly spaced between 0 and 1 on the sigmoid, and in fact, most don't.

I am assuming the code is correct because I had two versions (Python and R) to compare to and to double check dimensions, values, and how the calculations performed. Of course I could have missed some small detail somewhere but I have spent a considerable amount of time trying to find a possible bug without success. If you do find something I'd love to hear about it though.