Julia SVM Helper Functions

```
###############################
## by Qin Yu, Feb 2019
## using Julia 1.0.3
###############################

gaussian_kernel_matrix(𝑋, γ) =  # 𝑋's rows are input tensors.
    exp.(- γ * pairwise(SqEuclidean(), Matrix(𝑋), dims=1))
gaussian_kernel_matrix(𝑋, 𝑋_test, γ) =  # 𝑋's rows are input tensors.
    exp.(- γ * pairwise(SqEuclidean(), Matrix(𝑋_test), Matrix(𝑋), dims=1))

function extract_each_classes(𝑋_ALL, 𝒚_ALL)
    CLASSES = sort(unique(𝒚_ALL))
    NO_OF_CLASSES = length(CLASSES)
    L = [sum(𝒚_ALL .== class) for class in CLASSES]
    𝑋_EACH_train = [𝑋_ALL[:,:,𝒚_ALL .== class] for class in CLASSES]
    𝑋s = [hcat(transpose(reshape(𝑋_EACH_train[i], 28*28, L[i])), ones(L[i])) for i = 1:NO_OF_CLASSES]
    𝒚s = ones.(Int, L) .* CLASSES
    return L, 𝑋s, 𝒚s
end

function extract_all_test(𝑋_ALL, 𝒚_ALL)
    𝑋s = hcat(transpose(reshape(𝑋_ALL, 28*28, :)), ones(size(𝑋_ALL, 3)))
    return 𝑋s, 𝒚_ALL
end

function prepare_one_vs_rest(𝑋_ALL, 𝒚_ALL)
    𝑋 = hcat(transpose(reshape(𝑋_ALL, 28*28, :)), ones(size(𝑋_ALL, 3)))
    𝒚s = [(y -> y == pos_label ? 1 : -1).(𝒚_ALL) for pos_label = 0:9]
    return 𝑋, 𝒚s
end

function prepare_MNIST_data(class_pos, class_neg, γ)
    # Load train data
    𝑋_ALL_train, 𝒚_ALL_train = MNIST.traindata(Float32)

    NO_OF_NEGATIVE = sum(𝒚_ALL_train .== class_neg)
    NO_OF_POSITIVE = sum(𝒚_ALL_train .== class_pos)
    l = NO_OF_NEGATIVE + NO_OF_POSITIVE

    𝑋_neg_train = 𝑋_ALL_train[:,:,𝒚_ALL_train .== class_neg]
    𝑋_pos_train = 𝑋_ALL_train[:,:,𝒚_ALL_train .== class_pos]
    𝒚_neg_train = - ones(Int, NO_OF_NEGATIVE)
    𝒚_pos_train =   ones(Int, NO_OF_POSITIVE)

    randomisation = randperm(l)
    𝑋 = hcat(transpose(reshape(cat(𝑋_neg_train, 𝑋_pos_train; dims=3), 28*28, l)), ones(l))[randomisation,:]
    𝒚 = vcat(𝒚_neg_train, 𝒚_pos_train)[randomisation,:]

    gaussian_kernel_matrix(𝑋, γ) = exp.(- γ * pairwise(SqEuclidean(), Matrix(𝑋), dims=1))  # 𝑋's rows are input tensors.
    K = gaussian_kernel_matrix(𝑋, γ)

    # Load test data
    𝑋_ALL_test,  𝒚_ALL_test  = MNIST.testdata(Float32)

    NO_OF_NEGATIVE_test = sum(𝒚_ALL_test .== class_neg)
    NO_OF_POSITIVE_test = sum(𝒚_ALL_test .== class_pos)
    l_test = NO_OF_NEGATIVE_test + NO_OF_POSITIVE_test

    𝑋_neg_test = 𝑋_ALL_test[:,:,𝒚_ALL_test .== class_neg]
    𝑋_pos_test = 𝑋_ALL_test[:,:,𝒚_ALL_test .== class_pos]
    𝒚_neg_test = - ones(Int, NO_OF_NEGATIVE_test)
    𝒚_pos_test =   ones(Int, NO_OF_POSITIVE_test)

    𝑋_test = hcat(transpose(reshape(cat(𝑋_neg_test, 𝑋_pos_test; dims=3), 28*28, l_test)), ones(l_test))
    𝒚_test = vcat(𝒚_neg_test, 𝒚_pos_test)

    gaussian_kernel_matrix(𝑋, 𝑋_test, γ) = exp.(- γ * pairwise(SqEuclidean(), Matrix(𝑋_test), Matrix(𝑋), dims=1))
    K_test = gaussian_kernel_matrix(𝑋, 𝑋_test, γ)

    return (𝑋, 𝒚, K, l), (𝑋_test, 𝒚_test, K_test, l_test)
end

# Only as termination criterion for the whole process.
function monitor_kkt_condition(𝜶, 𝝈, 𝒚; accuracy=1e-2, 𝐶::Int32=Int32(1))
    𝑦𝑓𝑥 = 𝒚 .* 𝝈
    for (idx,α) in enumerate(𝜶)
        if α == 0 && 𝑦𝑓𝑥[idx] < 1
            return false
        elseif α == 𝐶 && 𝑦𝑓𝑥[idx] > 1
```

```
            return false
        elseif 0 < α < 𝐶 && !isapprox(𝑦𝑓𝑥[idx], 1; atol=accuracy)
            return false
        end
    end
    return true
end

function select_new_points(𝜶, 𝝈, 𝒚, 𝑀, support_vector_idx; accuracy=1e-2, 𝐶::Int32=Int32(1))
    𝑦𝑓𝑥 = 𝒚 .* 𝝈
    list_of_new_ids = Int[]
    list_of_new_min = Float64[]
    for (idx,α) in enumerate(𝜶)
        if (α == 0 && 𝑦𝑓𝑥[idx] < 1) || (α == 𝐶 && 𝑦𝑓𝑥[idx] > 1) || (0 < α < 𝐶 && !isapprox(𝑦𝑓𝑥[idx], 1; atol=accuracy))
            push!(list_of_new_ids, idx)
            push!(list_of_new_min, 𝑦𝑓𝑥[idx])
        end
    end
    list_of_new_ids = list_of_new_ids[sortperm(list_of_new_min)]
    setdiff!(list_of_new_ids, support_vector_idx)
    return list_of_new_ids
end
```

```
select_new_points (generic function with 1 method)
```

Julia SVM

```
##############################
## by Qin Yu, Apr 2019
## using Julia 1.1.0
##############################

using Revise, BenchmarkTools                           # Development
using JLD2, FileIO, MLDatasets                         # Data & IO
using LinearAlgebra, Distances, Random, Distributions  # Maths
using CUDAdrv, CUDAnative, CuArrays                    # GPU

MAX_MINI_BATCH_ID = 50

#################### Compute Adatron SGD in GPU: ####################
@inline sync_threads_and(predicate::Int32) = ccall("llvm.nvvm.barrier0.and", llvmcall, Int32, (Int32,), predicate)
@inline sync_threads_and(predicate::Bool) = ifelse(sync_threads_and(Int32(predicate)) !== Int32(0), true, false)

function kernel_soft_SGD_SVM(𝜶, 𝝈, K, 𝒚, l::Int32, 𝐶::Int32)
    j = (blockIdx().x-1) * blockDim().x + threadIdx().x
    can_stop = false
    δᵢ = @cuStaticSharedMem(Float32, 1)
    while !sync_threads_and(can_stop)
        last_α_j = 𝜶[j]
        # Adatron:
        for i = 1:l
            if j == i  # Online
                last_α = 𝜶[i]
                𝜇ᵢ = 1 / K[i,i]
                δᵢ[1] = 𝜇ᵢ * (1 - 𝒚[i] * 𝝈[i])
                𝜶[i] = 𝜶[i] + δᵢ[1]
                𝜶[i] < 0 && (𝜶[i] = 0; δᵢ[1] = 0 - last_α)
                𝜶[i] > 𝐶 && (𝜶[i] = 𝐶; δᵢ[1] = 𝐶 - last_α)
            end
            sync_threads()
            𝝈[j] += δᵢ[1] * 𝒚[i] * K[i,j]  # Parallel update
        end
        # Stopping criterion:
        can_stop = false
        isapprox(𝜶[j], last_α_j; atol=1e-4) && (can_stop = true)
    end
    return nothing
end

function optimise_working_set(𝜶, 𝝈, K, 𝒚; 𝐶::Int32=Int32(1))
    l = Int32(length(𝜶))
    cu_𝜶 = CuArray{Float32}(𝜶)
    cu_𝝈 = CuArray{Float32}(𝝈)
    cu_K = CuArray{Float32}(K)
    cu_𝒚 = CuArray{Float32}(𝒚)

    @cuda threads=l kernel_soft_SGD_SVM(cu_𝜶, cu_𝝈, cu_K, cu_𝒚, l, 𝐶)

    𝜶 = Array{Float32}(cu_𝜶)
    return 𝜶
end
```

```julia
function optimise_working_set_CPU(α, σ, K, y; C=1)
    l = length(α)
    while true
        last_α = copy(α)
        for i = 1:l
            μᵢ = 1 / K[i,i]
            δᵢ = μᵢ * (1 - y[i] * σ[i])
            α[i] = α[i] + δᵢ
            α[i] < 0 && (α[i] = 0; δᵢ = 0 - last_α[i])
            α[i] > C && (α[i] = C; δᵢ = C - last_α[i])
            σ .+= δᵢ * y[i] * K[i,:]
        end
        all(isapprox.(last_α, α; atol=1e-4)) && break
    end
    return α
end

function stochastic_decomposition_test(C, M, ACCURACY, l, y, K, l_test, y_test, K_test; usecpu=false)
    M_safe = div(M, 4) * 3
    M_rand = M - M_safe

    α = zeros(Float32, l)
    σ = zeros(Float32, l)

    selected_new_idx = []
    support_vector_idx, alpha0_vector_idx = collect(1:M), collect(M+1:l)
    working_set_counter = zeros(Int32, l)

    batch_id = 0
    while !monitor_kkt_condition(α, σ, y; C=C) && batch_id < MAX_MINI_BATCH_ID
        batch_id += 1

        current_working_set_counter = working_set_counter[support_vector_idx]
        sorted_support_vector_idx = support_vector_idx[sortperm(current_working_set_counter)]

        if length(support_vector_idx) >= M_safe
            no_of_new_idx = min(length(selected_new_idx), M_rand)
        else
            no_of_new_idx = min(length(selected_new_idx), M-length(support_vector_idx))
        end

        no_of_old_idx = length(support_vector_idx)
        no_of_exceeds = no_of_new_idx + no_of_old_idx - M
        if no_of_exceeds > 0
            abandoned_support_vector_idx = sorted_support_vector_idx[end+1-no_of_exceeds:end]
            setdiff!(support_vector_idx, abandoned_support_vector_idx)
            union!(alpha0_vector_idx, abandoned_support_vector_idx)
        end
        selected_new_idx = selected_new_idx[1:no_of_new_idx]
        union!(support_vector_idx, selected_new_idx)
        setdiff!(alpha0_vector_idx, selected_new_idx)

        working_set_counter[support_vector_idx] .+= 1
        α_subset, σ_subset = α[support_vector_idx], σ[support_vector_idx]
        y_subset, K_subset = y[support_vector_idx], K[support_vector_idx,support_vector_idx]
        if usecpu
            α[support_vector_idx] = optimise_working_set_CPU(α_subset, σ_subset, K_subset, y_subset; C=C)
        else
            α[support_vector_idx] = optimise_working_set(α_subset, σ_subset, K_subset, y_subset; C=C)
        end

        σ = K * (α .* y)

        for (i_index,i) in enumerate(support_vector_idx)
            if α[i] == 0
                deleteat!(support_vector_idx, i_index)
                push!(alpha0_vector_idx, i)
            end
        end

        selected_new_idx = select_new_points(α, σ, y, M, support_vector_idx; accuracy=ACCURACY, C=C)
    end

    ŷ      = sign.(σ)
    ŷ_test = sign.(K_test * (α .* y))
    error_rate      = count(ŷ .!= y) / l * 100
    error_rate_test = count(ŷ_test .!= y_test) / l_test * 100
    return α, error_rate, error_rate_test, monitor_kkt_condition(α, σ, y; C=C)
end

function stochastic_decomposition(C, M, ACCURACY, l, y, K; usecpu=false)
    M_safe = div(M, 4) * 3
```

```
    𝑀_rand = 𝑀 - 𝑀_safe

    α = zeros(Float32, l)
    σ = zeros(Float32, l)

    selected_new_idx = []
    support_vector_idx, alpha0_vector_idx = collect(1:𝑀), collect(𝑀+1:l)
    working_set_counter = zeros(Int32, l)

    batch_id = 0
    while !monitor_kkt_condition(α, σ, 𝑦; accuracy=ACCURACY, 𝐶=𝐶) && batch_id < MAX_MINI_BATCH_ID
        batch_id += 1

        current_working_set_counter = working_set_counter[support_vector_idx]
        sorted_support_vector_idx = support_vector_idx[sortperm(current_working_set_counter)]

        if length(support_vector_idx) >= 𝑀_safe
            no_of_new_idx = min(length(selected_new_idx), 𝑀_rand)
        else
            no_of_new_idx = min(length(selected_new_idx), 𝑀-length(support_vector_idx))
        end

        no_of_old_idx = length(support_vector_idx)
        no_of_exceeds = no_of_new_idx + no_of_old_idx - 𝑀
        if no_of_exceeds > 0
            abandoned_support_vector_idx = sorted_support_vector_idx[end+1-no_of_exceeds:end]
            setdiff!(support_vector_idx, abandoned_support_vector_idx)
            union!(alpha0_vector_idx, abandoned_support_vector_idx)
        end
        selected_new_idx = selected_new_idx[1:no_of_new_idx]
        union!(support_vector_idx, selected_new_idx)
        setdiff!(alpha0_vector_idx, selected_new_idx)

        working_set_counter[support_vector_idx] .+= 1
        α_subset, σ_subset = α[support_vector_idx], σ[support_vector_idx]
        𝑦_subset, K_subset = 𝑦[support_vector_idx], K[support_vector_idx,support_vector_idx]
        if usecpu
            println("using CPU")
            α[support_vector_idx] = optimise_working_set_CPU(α_subset, σ_subset, K_subset, 𝑦_subset; 𝐶=𝐶)
        else
            α[support_vector_idx] = optimise_working_set(α_subset, σ_subset, K_subset, 𝑦_subset; 𝐶=𝐶)
        end
        σ = K * (α .* 𝑦)

        for (i_index,i) in enumerate(support_vector_idx)
            if α[i] == 0
                deleteat!(support_vector_idx, i_index)
                push!(alpha0_vector_idx, i)
            end
        end

        selected_new_idx = select_new_points(α, σ, 𝑦, 𝑀, support_vector_idx; accuracy=ACCURACY, 𝐶=𝐶)
    end

    𝑦̂ = sign.(σ)
    error_rate = count(𝑦̂ .!= 𝑦) / l * 100
    return α, error_rate, monitor_kkt_condition(α, σ, 𝑦; accuracy=ACCURACY, 𝐶=𝐶)
end
```

```
stochastic_decomposition (generic function with 1 method)
```

MNIST Training and Testing

```
##############################
## by Qin Yu, Apr 2019
## using Julia 1.1.0
##############################

#################### MNIST One-vs-One Multi-class ####################
function MNIST_train(L, L_test, 𝑋s, 𝑦s, 𝐶, 𝑀, ACCURACY, γ; usecpu=false)
    time_K_spent = time_GPU_spent = time_total_spent = 0
    time_total_start = time()

    randomisation_list = Array{Int32,1}[]
    α_list = Array{Float32,1}[]
    error_list = Float64[]
    error_list_test = Float64[]
    finished_list = Bool[]
    time_ivj_list = Float64[]

    for ci = 1:10, cj = 1:10
```

```
        ci >= cj && continue
        print("$(ci-1)-vs-$(cj-1) - ")

        time_K_start = time()
        l = L[ci] + L[cj]
        randomisation = randperm(l)
        𝑦 = vcat(ones(Int32, size(𝑦s[ci])), -ones(Int32, size(𝑦s[cj])))[randomisation,:]
        𝑋 = vcat(𝑋s[ci], 𝑋s[cj])[randomisation,:]
        K = gaussian_kernel_matrix(𝑋, γ)
        time_K_spent += time() - time_K_start

        # l_test = L_test[ci] + L_test[cj]
        # randomisation_test = randperm(l_test)
        # 𝑦_test = vcat(ones(Int32, size(𝑦s_test[ci])), -ones(Int32, size(𝑦s_test[cj])))[randomisation_test,:]
        # 𝑋_test = vcat(𝑋s_test[ci], 𝑋s_test[cj])[randomisation_test,:]
        # K_test = gaussian_kernel_matrix(𝑋, 𝑋_test, γ)

        time_GPU_start = time()
        𝜶, error_rate, finished = stochastic_decomposition(𝐶, 𝑀, ACCURACY, l, 𝒚, K; usecpu=usecpu)
        # 𝜶, error_rate, error_rate_test, finished =
        #     stochastic_decomposition(𝐶, 𝑀, l, 𝒚, K, l_test, 𝒚_test, K_test)
        time_GPU_spent_this = time() - time_GPU_start
        push!(time_ivj_list, time_GPU_spent_this)
        time_GPU_spent += time_GPU_spent_this

        println("error rate = $error_rate% ; finished = $finished ; time = $time_GPU_spent_this")
        # println("error rate = $error_rate% ; test error rate = $error_rate_test% ;
        #     finished = $finished ; time = $time_GPU_spent_this")

        push!(randomisation_list, randomisation)
        push!(𝜶_list, 𝜶)
        push!(error_list, error_rate)
        # push!(error_list_test, error_rate_test)
        push!(finished_list, finished)
    end

    time_total_spent = time() - time_total_start
    return ([time_K_spent, time_GPU_spent, time_total_spent],
            randomisation_list, 𝜶_list, error_list, error_list_test, finished_list, time_ivj_list)
end

function MNIST_test_slave(𝑋_ALL, 𝑦_ALL, 𝑋s, 𝑦s, 𝜶_list, randomisation_list, γ)
    i = 0
    𝑦̂_test_list = Array{Int}[]
    for ci = 1:10, cj = 1:10
        ci >= cj && continue
        print("$(ci-1)-vs-$(cj-1) - ")
        i += 1

        𝑋_test, 𝑦_test = extract_all_test(𝑋_ALL, 𝑦_ALL)
        K_test = gaussian_kernel_matrix(vcat(𝑋s[ci], 𝑋s[cj])[randomisation_list[i],:], 𝑋_test, γ)
        𝑦_true = vcat(ones(Int32, size(𝑦s[ci])), -ones(Int32, size(𝑦s[cj])))[randomisation_list[i],:]
        𝝈_test = K_test * (𝜶_list[i] .* 𝑦_true)
        𝑦̂_test = (y -> y > 0 ? ci-1 : cj-1).(𝝈_test)

        push!(𝑦̂_test_list, 𝑦̂_test)
    end
    return 𝑦̂_test_list
end

function MNIST_test(𝑦_test, 𝑦̂_test_list)
    𝑦̂_test_matrix = hcat(𝑦̂_test_list...)
    𝑦̂_test = mapslices(mode, 𝑦̂_test_matrix; dims=2)
    error_rate_test = count(𝑦̂_test .!= 𝑦_test) / length(𝑦_test) * 100
    correct_rate_test = 100 - error_rate_test  # First run: 99.1%
    return 𝑦̂_test, error_rate_test, correct_rate_test
end

function run(𝐶::Int32=Int32(1),     # Penalty
             𝑀::Int=512,           # Max size of minibatch
             ACCURACY=1e-2,        # GPU accuracy = 0.01 * ACCURACY
             γ=0.015;              # Gaussian kernel parameter
             usecpu=false)
    𝑋_ALL, 𝑦_ALL = MNIST.traindata(Float32)
    𝑋_ALL_test, 𝑦_ALL_test = MNIST.testdata(Float32)
    L, 𝑋s, 𝑦s = extract_each_classes(𝑋_ALL, 𝑦_ALL)
    L_test, 𝑋s_test, 𝑦s_test = extract_each_classes(𝑋_ALL_test,  𝑦_ALL_test)

    @time time_list, randomisation_list, 𝜶_list, error_list, error_list_test, finished_list, time_ivj_list =
        MNIST_train(L, L_test, 𝑋s, 𝑦s, 𝐶, 𝑀, ACCURACY, γ; usecpu=usecpu)
    @save "1v1.jld2" time_list, randomisation_list, 𝜶_list, error_list, error_list_test, finished_list, time_ivj_list
    @time 𝑦̂_test_list = MNIST_test_slave(𝑋_ALL, 𝑦_ALL, 𝑋s, 𝑦s, 𝜶_list, randomisation_list, γ)
    @time 𝑦̂_test, error_rate_test = MNIST_test(𝑦_ALL, 𝑦̂_test_list)
```

```
    println(error_rate_test)
end
```

```
run (generic function with 5 methods)
```

Run SVM

```
run()            # 1256.897736s ≈ 20.95min
```

```
0-vs-1 - error rate = 0.007895775759968417% ; finished = true ; time = 11.731913089752197
0-vs-2 - error rate = 0.042083999663328% ; finished = true ; time = 12.422368049621582
0-vs-3 - error rate = 0.024888003982080638% ; finished = true ; time = 10.846972942352295
0-vs-4 - error rate = 0.06799830004249893% ; finished = true ; time = 7.246622085571289
0-vs-5 - error rate = 0.07052186177715092% ; finished = true ; time = 18.025906801223755
0-vs-6 - error rate = 0.10978802466007939% ; finished = true ; time = 10.650660037994385
0-vs-7 - error rate = 0.016409583196586804% ; finished = true ; time = 7.11625599861145
0-vs-8 - error rate = 0.08493290300662477% ; finished = true ; time = 11.790964126586914
0-vs-9 - error rate = 0.07580862533692723% ; finished = true ; time = 7.729828834533691
1-vs-2 - error rate = 0.13385826771653542% ; finished = true ; time = 16.983326196670532
1-vs-3 - error rate = 0.06991377301328362% ; finished = true ; time = 11.7693190574646
1-vs-4 - error rate = 0.14303877940241577% ; finished = true ; time = 8.491616010665894
1-vs-5 - error rate = 0.04932993504891885% ; finished = true ; time = 11.43572211265564
1-vs-6 - error rate = 0.023696682464454978% ; finished = true ; time = 22.854277849197388
1-vs-7 - error rate = 0.261397708925963% ; finished = true ; time = 12.761433839797974
1-vs-8 - error rate = 0.1508774716112126% ; finished = true ; time = 25.24915885925293
1-vs-9 - error rate = 0.11031439602868175% ; finished = true ; time = 7.722901821136475
2-vs-3 - error rate = 0.2233435354454463% ; finished = true ; time = 51.78821110725403
2-vs-4 - error rate = 0.11864406779661016% ; finished = true ; time = 24.21775197982788
2-vs-5 - error rate = 0.043940592319184464% ; finished = true ; time = 24.209162950515747
2-vs-6 - error rate = 0.0421017177500842% ; finished = true ; time = 26.605338096618652
2-vs-7 - error rate = 0.29452671193651314% ; finished = true ; time = 31.75123906135559
2-vs-8 - error rate = 0.20323482089931408% ; finished = true ; time = 57.46496510505676
2-vs-9 - error rate = 0.07558578987150416% ; finished = true ; time = 15.855940103530884
3-vs-4 - error rate = 0.05011275369581559% ; finished = true ; time = 15.090398073196411
3-vs-5 - error rate = 0.3808864265927978% ; finished = true ; time = 66.38876104354858
3-vs-6 - error rate = 0.008299443937256203% ; finished = true ; time = 14.185173988342285
3-vs-7 - error rate = 0.18554372378186512% ; finished = true ; time = 28.159248113632202
3-vs-8 - error rate = 0.35052578868302453% ; finished = true ; time = 55.56105613708496
3-vs-9 - error rate = 0.25662251655629137% ; finished = true ; time = 22.970326900482178
4-vs-5 - error rate = 0.03551451655864335% ; finished = true ; time = 19.871274948120117
4-vs-6 - error rate = 0.13605442176870747% ; finished = true ; time = 15.57249402999878
4-vs-7 - error rate = 0.21475179648137438% ; finished = true ; time = 42.69997715950012
4-vs-8 - error rate = 0.09407337723424271% ; finished = true ; time = 19.575289964675903
4-vs-9 - error rate = 0.5003816470189127% ; finished = true ; time = 73.25518202781677
5-vs-6 - error rate = 0.17638239703677575% ; finished = true ; time = 26.81020998954773
5-vs-7 - error rate = 0.06845798391237377% ; finished = true ; time = 20.46792197227478
5-vs-8 - error rate = 0.17743080198722497% ; finished = true ; time = 53.68216514587402
5-vs-9 - error rate = 0.12313104661389623% ; finished = true ; time = 23.383502960205078
6-vs-7 - error rate = 0.008208158909956496% ; finished = true ; time = 6.258177995681763
6-vs-8 - error rate = 0.05098139179199593% ; finished = true ; time = 16.659406185150146
6-vs-9 - error rate = 0.02528018875874273% ; finished = true ; time = 7.46078896522522
7-vs-8 - error rate = 0.115549686365137% ; finished = true ; time = 18.93709897994995
7-vs-9 - error rate = 0.4830522351400033% ; finished = true ; time = 76.98206496238708
8-vs-9 - error rate = 0.1440677966101695% ; finished = true ; time = 44.56295299530029
Total training time 1278.997176 seconds (19.93 M allocations: 159.586 GiB, 1.11% gc time)


Final error rate 0.895%
```