

```

using Plots
using LinearAlgebra
using JuMP
using Gurobi
using MLDatasets
import Statistics
import Random

"""
    solve_LR_coef(X,y,b; LIMIT)
Logistic Regression
Computes parameters b for logistic regression using the Newton-Rhapson Algorithm.
X must have 1's appended as first column. y must have 1,0 encoding.
"""
function solve_LR_coef(X,y,b; LIMIT = 1000)
    n = size(X,1)
    # Probability function
    prob(x, β) = 1 / (1 + exp(-(dot(x,β) )))
    # Gradient of Loss Function
    ∇l(β,X,y,n) = sum( X[i,:] * (y[i] - prob(X[i,:], β)) for i = 1:n)
    # Hessian of Loss Function (not used)
    # Hl(β,X,n) = -sum( X[i,:] * transpose(X[i,:]) * prob(X[i,:], β)*(1-prob(X[i,:], β)) for
    i = 1:n )

    global counter = 0
    tol = 1.e-6
    # Gradient descent
    while norm(∇l(b,X,y,n)) > tol && counter < LIMIT
        W = zeros(n,n)
        P = zeros(n)
        for i = 1:n
            W[i,i] = prob(X[i,:], b)*(1 - prob(X[i,:], b))
            P[i] = prob(X[i,:], b)
        end
        approx = W \ (y-P)
        z = X*b + approx
        # z = X*b + inv(W)*(y-P) # may not be invertible
        b = inv((X'*W*X)) * X'*W*z
        global counter += 1
    end

    println("Number of iterations: $counter.")
    if counter == LIMIT
        println("Max iterations reached.")
    end

    return b
end

```

```

"""
    kernel_ridge(X,y,λ,K)
Kernel Ridge Regression
Computes parameters  $\alpha$  for the kernel ridge estimator with kernel function K and
hyperparameter  $\lambda$ .
"""
function kernel_ridge(X,y,λ,K)
    n = length(y)
    k = zeros(n,n)
    for i = 1:n
        for j = i:n
            k[i,j] = K(X[i,:],X[j,:])
            k[j,i] = k[i,j]
            # println("$k[i,j]")
        end
    end
    # Solution  $\alpha$  to min (norm(y-k*α))^2 + λ α'*k*α
    α = (k + λ*I) \ y

    #  $\hat{f}(x^*) = \sum( K(x^*,X[i,:]) * \alpha[i] )$ 
    return α
end

"""
    prox_grad_desc(X,y,β,λ; LIMIT)
Proximal Gradient Descent
Computes parameters  $\beta$  given initial guess for least squares regression with  $\ell_1$  penalty.
Has hyperparameter  $\lambda$ .
"""
function prox_grad_desc(X,y,β,λ; LIMIT=1000)
    # Soft-thresholding function
    S(y,l) = begin
        if abs(y) <= l
            return 0
        elseif y > l
            return y - l
        else
            return y + l
        end
    end

    # Solving min 1/2*norm(Y-Xβ,2)^2 + λ*norm(β,1)
    h = 1 / max(eigvals(X'X)...) # learning rate
    ∇L(β) = -X'*(y-X*β)
    tol = 1.e-6
    counter = 0
    while norm(∇L(β)) > tol && counter < LIMIT
        β = S.(β - h*∇L(β), λ*h)
        counter += 1
    end
end

```

```

        # if counter % 10 == 0
        #     println("Iteration $counter, gradient norm $(norm(∇L(β)))")
        # end
    end

    println("Number of iterations: $counter.")
    if counter == LIMIT
        println("Max iterations reached.")
    end

    return β
end

"""
    elastic_net(X,y,β,λ,α; LIMIT)
Elastic Net
Computes parameters β given initial guess for least squares regression with elastic net
penalty.
Uses soft-thresholding function update derived in Homework 2. Hyperparameters are λ and α.
"""
function elastic_net(X,y,β,λ,α; LIMIT=1000)
    # Soft-thresholding function
    S(y,l) = begin
        if abs(y) <= l
            return 0
        elseif y > l
            return y - l
        else
            return y + l
        end
    end

    # Solving min 1/2*norm(Y-Xβ,2)^2 + λ*(α*norm(β,1) + (1-α)*norm(β,2)^2)
    η = 1 / max(eigvals(X'X)...) # learning rate
    ∇L(β) = -X'*(y-X*β) + 2λ*(1-α)*β
    tol = 1.e-6
    counter = 0
    while norm(∇L(β)) > tol && counter < LIMIT
        β = S.(β - η*∇L(β), λ*η*α)
        counter += 1
        # if counter % 10 == 0
        #     println("Iteration $counter, gradient norm $(norm(∇L(β)))")
        # end
    end

    println("Number of iterations: $counter.")
    if counter == LIMIT
        println("Max iterations reached.")
    end
end

```

```

        return  $\beta$ 
    end

"""
    SVM(X,y,C)
Support Vector Machines
Computes parameters  $\beta_0$ ,  $\beta$  and the margins  $\xi$  for linear SVMs. Uses Gurobi to solve the
optimization problem. Has hyperparameter C. y must have -1,1 encoding.
"""
function SVM(X,y,C)
    n,p = size(X)
    model = JuMP.Model(JuMP.optimizer_with_attributes(Gurobi.Optimizer, "MIPGap" => .01,
"TimeLimit" => 180))
    JuMP.@variable(model,  $\beta_0$ )
    JuMP.@variable(model,  $\beta[1:p]$ )
    JuMP.@variable(model,  $\xi[1:n] \geq 0$ )
    JuMP.@objective(model, Min,  $\beta' \cdot \beta$ )
    for i = 1:n
        JuMP.@constraint(model,  $y[i] \cdot (X[i,:] \cdot \beta + \beta_0) \geq 1 - \xi[i]$ )
    end
    JuMP.@constraint(model, sum( $\xi[i]$  for i = 1:n) <= C)

    JuMP.optimize!(model)

    return value( $\beta_0$ ), value.( $\beta$ ), value.( $\xi$ )
end

"""
    kernel_SVM(X,y,C,K)
Kernel Support Vector Machines
Computes parameters  $\alpha$  for Kernel SVMs with kernel function K and hyperparameter C. Uses
Gurobi
to solve the optimization problem. y must have -1,1 encoding.
"""
function kernel_SVM(X,y,C,K)
    n = size(X,1)
    k = zeros(n,n)
    for i = 1:n
        for j = i:n
            k[i,j] = K(X[i,:],X[j,:])
            k[j,i] = k[i,j]
            # println(" $\$(k[i,j])$ ")
        end
    end

    model = JuMP.Model(JuMP.optimizer_with_attributes(Gurobi.Optimizer, "NonConvex" => 2,
"MIPGap" => .01, "TimeLimit" => 180))

```

```

JuMP.@variable(model, 0 <= α[1:n] <= C)
JuMP.@objective(model,
    Max,
    sum(α[i] for i = 1:n) - 1/2 * sum( sum( α[i]*α[j]*y[i]*y[j]*k[i,j] for j
= 1:n ) for i = 1:n)
    )
JuMP.@constraint(model, sum(α[i]*y[i] for i = 1:n) == 0)

JuMP.optimize!(model)

return value.(α)
end

"""
    bootstrapper(X,y,B)
Bootstrapping Procedure
Returns B bootstrapped samples of input data X and y. Samples are stored in an array. The
indices used and not used in each sample are also returned for out-of-bag error calculation.
"""
function bootstrapper(X,y,B)
    n,p = size(X)
    b_samples = []
    ind_used = []
    ind_not_used = []
    for _ = 1:B
        X_b = zeros(n,p)
        y_b = zeros(n)
        ind_used_b = Set{Int{}}()
        for i=1:n
            ind = rand(1:n)
            X_b[i,:] = X[ind,:]
            y_b[i] = y[ind]
            push!(ind_used_b, ind)
        end
        ind_all = Set{Int{}}(1:n)
        ind_not_used_b = setdiff(ind_all,ind_used_b)
        push!(b_samples, (X_b,y_b))
        push!(ind_used, ind_used_b)
        push!(ind_not_used, ind_not_used_b)
    end

    return b_samples, ind_used, ind_not_used
end

"""
    train_neural_network(X,y,num_hidden_layers,size_hidden_layers;h,activation,problem_type,n
um_classes,num_epochs)
Initiate Neural Network

```

Computes weight matrices  $W$  and biases  $b$  and prints the training error.  $W$  and  $b$  are initialized in this function. Activation options are 'sigmoid' and 'ReLU'. Problem types are 'classification' and 'regression'.

Other parameters settings are number of hidden layers, number of neurons per hidden layer, number of epochs, number of classes (1 if regression), and learning rate  $h$ .

```
"""
function train_neural_network(X,y,num_hidden_layers,size_hidden_layers;h=0.1,
activation="sigmoid",problem_type="classification",num_classes=1,num_epochs=1)
    # Create activation functions
    if activation == "sigmoid"
         $\sigma = (z) \rightarrow 1 / (1+\exp(-z))$ 
         $D\sigma = (z) \rightarrow \sigma(z)^2 * \exp(-z)$     # gradient with respect to z
        println("Sigmoid used.")
    elseif activation == "ReLU"
         $\sigma = (z) \rightarrow \max(z,0)$ 
         $D\sigma = (z) \rightarrow \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{else} \end{cases}$ 
        return 1.0
        else
            return 0.0
        end
    end
    println("ReLU used.")
end

# Create appropriate loss function
if problem_type == "classification"
    y_indicator = []
    classes = vcat(collect(1:9), 0)
    for i = 1:length(y)
        push!(y_indicator, map(k -> y[i] == k, classes))
    end

    L = (y_indicator,a_L_prob) -> -sum(y_indicator[k] * log(a_L_prob[k]) for
k=1:num_classes)
    DL = (y_indicator,a_L_prob) -> [-y_indicator[k] / a_L_prob[k] for k=1:num_classes]
    # These are gonna return vectors
    f = (z) -> [exp(z[j]) / sum(exp(z[k]) for k=1:num_classes) for j=1:num_classes]
    Df = (z) -> [(sum(exp(z[k]) for k=1:num_classes) - exp(z[j]))*exp(z[j]) /
(sum(exp(z[k]) for k=1:num_classes))^2 for j=1:num_classes]
elseif problem_type == "regression"
    L = (y,a_L) -> norm(y - a_L)^2
    DL = (y,a_L) -> 2*(a_L - y)
    f = (z) -> z
    Df = (z) -> 1
end

# Total number of layers
```

```

LL = num_hidden_layers + 2
# Stepsize
 $\eta$  = h

# To store weighted inputs
z = map(_ -> zeros(size_hidden_layers), 1:num_hidden_layers)
pushfirst!(z, zeros(size(X,2)))
push!(z, zeros(num_classes))
# To store activations
a = map(_ -> zeros(size_hidden_layers), 1:num_hidden_layers)
pushfirst!(a, zeros(size(X,2)))
push!(a, zeros(num_classes))
# To store errors
 $\delta$  = map(_ -> zeros(size_hidden_layers), 1:num_hidden_layers)
pushfirst!( $\delta$ , zeros(size(X,2)))
push!( $\delta$ , zeros(num_classes))

# Create biases
b = map(_ -> rand(size_hidden_layers), 1:num_hidden_layers)
pushfirst!(b, rand(size(X,2)))
push!(b, rand(num_classes))
# Create weight parameters
W = [Matrix{Float64}(I(size(X,2)))] # store identity for sake
push!(W, rand(size_hidden_layers, size(X,2)) ) # W[2] = weights from layer 1 to layer 2
for _ = 1:(num_hidden_layers-1)
    push!(W, rand(size_hidden_layers, size_hidden_layers))
end
push!(W, rand(num_classes, size_hidden_layers))

for _ = 1:num_epochs
    for i = 1:size(X,1)
        # Begin forward pass
        z[1] = X[i,:] # identical transformation
        a[1] = Vector{Float64}( $\sigma$ .(z[1]))
        for l = 2:(LL-1) # LL and LL-1
            z[l] = W[l]*a[l-1] + b[l]
            a[l] = Vector{Float64}( $\sigma$ .(z[l]))
        end
        z[LL] = W[LL]*a[LL-1] + b[LL]
        # a[LL] = f(z[LL]) # Regression requires a broadcasting with a . while
classification does not

        # Begin backward pass
        if problem_type == "classification"
            #  $\delta$ [LL] = DL(y_indicator[i], a[LL]) .* D $\sigma$ .(z[LL])
            a[LL] = f(z[LL])
             $\delta$ [LL] = DL(y_indicator[i], a[LL]) .* Df(z[LL])
        elseif problem_type == "regression"
            #  $\delta$ [LL] = DL(y[i], a[LL][1]) .* D $\sigma$ .(z[LL])
            a[LL] = f.(z[LL])

```

```

         $\delta[LL] = DL(y[i], a[LL][1]) .* Df.(z[LL])$ 
    end

    # Compute weighted errors
    for l = (LL-1):-1:2
         $\delta[l] = W[l+1]' * \delta[l+1] .* \text{Vector}\{\text{Float64}\}(D\sigma.(z[l]))$ 
    end

    # Update weights and biases
    for l = 2:LL
         $W[l] = W[l] - \eta * \delta[l] * a[l-1]'$ 
         $b[l] = b[l] - \eta * \delta[l]$ 
    end
end

end

# for l=2:LL
#     println("W_$l is:\n$(W[l])")
#     println("b_$l is:\n$(b[l])")
# end

function f_predictor(x)
    z[1] = x    # identical transformation
    a[1] = Vector{Float64}( $\sigma.(z[1])$ )
    for l = 2:(LL-1)
         $z[l] = W[l]*a[l-1] + b[l]$ 
        a[l] = Vector{Float64}( $\sigma.(z[l])$ )
    end
     $z[LL] = W[LL]*a[LL-1] + b[LL]$ 

    #println("$a[L]")

    if problem_type == "classification"
        a[LL] = f(z[LL])
        y_pred = argmax(a[LL])
        if y_pred == 10
            y_pred = 0
        end
        return y_pred, a[LL]
    elseif problem_type == "regression"
        a[LL] = f.(z[LL])
        return a[LL][1] # if regression, final layer has 1 output
    end
end

if problem_type == "classification"
    # loss = [L(y_indicator[i], f_predictor(X[i,:])) for i = 1:size(X,1)]
    y_pred_vec = zeros(size(X,1))
    a_L_vec = map(k -> zeros(num_classes), 1:size(X,1))
    for i = 1:size(X,1)

```



```

        y_pred_vec[i], a_L_vec[i] = f_predictor(X[i,:])
    end
    total_loss = sum([L(y_indicator[i], a_L_vec[i]) for i = 1:size(X,1)])
    println("Total loss is $total_loss.")
    ave_loss = Statistics.mean([L(y_indicator[i], a_L_vec[i]) for i = 1:size(X,1)])
    println("Average loss $ave_loss out of $(size(X,1)) training images.")

    # Compute misclassification
    misclass = 0
    for i = 1:size(X,1)
        if y_pred_vec[i] != y[i]
            misclass += 1
        end
    end
    println("$misclass of $(size(X,1)) training images misclassified.")
    return W,b
elseif problem_type == "regression"
    ave_loss = Statistics.mean([L(y[i], f_predictor(X[i,:])) for i = 1:size(X,1)])
    println("Average loss $ave_loss out of $(size(X,1)) training images.")
    return W,b
end

end

"""
    update_neural_network(X,y,W,b,num_hidden_layers,size_hidden_layers;h,activation,problem_t
ype,num_classes,num_epochs)
Update Neural Network
Computes weight matrices W and biases b and prints the training error. W and b are passed as
arguments in this
function. The network settings should be the same as those passed when creating W and b.
"""
function
update_neural_network(X,y,W,b,num_hidden_layers,size_hidden_layers;h=0.1,activation="sigmoid"
,problem_type="classification",num_classes=1,num_epochs=1)
    # Create activation functions
    if activation == "sigmoid"
         $\sigma = (z) \rightarrow 1 / (1 + \exp(-z))$ 
         $D\sigma = (z) \rightarrow \sigma(z)^2 * \exp(-z)$     # gradient with respect to z
        println("Sigmoid used.")
    elseif activation == "ReLU"
         $\sigma = (z) \rightarrow \max(z, 0)$ 
         $D\sigma = (z) \rightarrow \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{else} \end{cases}$ 
        if z >= 0.0
            return 1.0
        else
            return 0.0
        end
    end
    end
    println("ReLU used.")

```

```

end

# Create appropriate loss function
if problem_type == "classification"
    y_indicator = []
    classes = vcat(collect(1:9), 0)
    for i = 1:length(y)
        push!(y_indicator, map(k -> y[i] == k, classes))
    end

    L = (y_indicator, a_L_prob) -> -sum(y_indicator[k] * log(a_L_prob[k]) for
k=1:num_classes)
    DL = (y_indicator, a_L_prob) -> [-y_indicator[k] / a_L_prob[k] for k=1:num_classes]
    # These are gonna return vectors
    f = (z) -> [exp(z[j]) / sum(exp(z[k]) for k=1:num_classes) for j=1:num_classes]
    Df = (z) -> [(sum(exp(z[k]) for k=1:num_classes) - exp(z[j]))*exp(z[j]) /
(sum(exp(z[k]) for k=1:num_classes))^2 for j=1:num_classes]
elseif problem_type == "regression"
    L = (y, a_L) -> norm(y - a_L)^2
    DL = (y, a_L) -> 2*(a_L - y)
    f = (z) -> z
    Df = (z) -> 1
end

# Total number of layers
LL = num_hidden_layers + 2
# Stepsize
 $\eta$  = h

# To store weighted inputs
z = map(_ -> zeros(size_hidden_layers), 1:num_hidden_layers)
pushfirst!(z, zeros(size(X,2)))
push!(z, zeros(num_classes))
# To store activations
a = map(_ -> zeros(size_hidden_layers), 1:num_hidden_layers)
pushfirst!(a, zeros(size(X,2)))
push!(a, zeros(num_classes))
# To store errors
 $\delta$  = map(_ -> zeros(size_hidden_layers), 1:num_hidden_layers)
pushfirst!( $\delta$ , zeros(size(X,2)))
push!( $\delta$ , zeros(num_classes))

for _ = 1:num_epochs
    for i = 1:size(X,1)
        # Begin forward pass
        z[1] = X[i,:] # identical transformation
        a[1] = Vector{Float64}( $\sigma$ .(z[1]))
        for l = 2:(LL-1) # LL and LL-1
            z[l] = W[l]*a[l-1] + b[l]
            a[l] = Vector{Float64}( $\sigma$ .(z[l]))
        end
    end
end

```

```

        end
        z[LL] = W[LL]*a[LL-1] + b[LL]
        # a[LL] = f(z[LL])    # Regression requires a broadcasting with a . while
classification does not

        # Begin backward pass
        if problem_type == "classification"
            #  $\delta[LL] = DL(y\_indicator[i], a[LL]) \cdot D\sigma(z[LL])$ 
            a[LL] = f(z[LL])
             $\delta[LL] = DL(y\_indicator[i], a[LL]) \cdot Df(z[LL])$ 
        elseif problem_type == "regression"
            #  $\delta[LL] = DL(y[i], a[LL][1]) \cdot D\sigma(z[LL])$ 
            a[LL] = f.(z[LL])
             $\delta[LL] = DL(y[i], a[LL][1]) \cdot Df.(z[LL])$ 
        end

        # Compute weighted errors
        for l = (LL-1):-1:2
             $\delta[l] = W[l+1]' \cdot \delta[l+1] \cdot \text{Vector}\{\text{Float64}\}(D\sigma.(z[l]))$ 
        end

        # Update weights and biases
        for l = 2:LL
            W[l] = W[l] -  $\eta \cdot \delta[l] \cdot a[l-1]'$ 
            b[l] = b[l] -  $\eta \cdot \delta[l]$ 
        end
    end
end

# for l=2:LL
#     println("W_$l is:\n$(W[l])")
#     println("b_$l is:\n$(b[l])")
# end

function f_predictor(x)
    z[1] = x    # identical transformation
    a[1] = Vector{Float64}( $\sigma.(z[1])$ )
    for l = 2:(LL-1)
        z[l] = W[l]*a[l-1] + b[l]
        a[l] = Vector{Float64}( $\sigma.(z[l])$ )
    end
    z[LL] = W[LL]*a[LL-1] + b[LL]

    #println("$a[L]")

    if problem_type == "classification"
        a[LL] = f(z[LL])
        y_pred = argmax(a[LL])
        if y_pred == 10
            y_pred = 0
        end
    end
end

```

```

        end
        return y_pred, a[LL]
    elseif problem_type == "regression"
        a[LL] = f.(z[LL])
        return a[LL][1] # if regression, final layer has 1 output
    end
end

if problem_type == "classification"
    # loss = [L(y_indicator[i], f_predictor(X[i,:])) for i = 1:size(X,1)]
    y_pred_vec = zeros(size(X,1))
    a_L_vec = map(k -> zeros(num_classes), 1:size(X,1))
    for i = 1:size(X,1)
        y_pred_vec[i], a_L_vec[i] = f_predictor(X[i,:])
    end
    total_loss = sum([L(y_indicator[i], a_L_vec[i]) for i = 1:size(X,1)])
    println("Total loss is $total_loss.")
    ave_loss = Statistics.mean([L(y_indicator[i], a_L_vec[i]) for i = 1:size(X,1)])
    println("Average loss $ave_loss out of $(size(X,1)) training images.")

    # Compute misclassification
    misclass = 0
    for i = 1:size(X,1)
        if y_pred_vec[i] != y[i]
            misclass += 1
        end
    end
    println("$misclass of $(size(X,1)) training images misclassified.")
    return W,b
elseif problem_type == "regression"
    ave_loss = Statistics.mean([L(y[i], f_predictor(X[i,:])) for i = 1:size(X,1)])
    println("Average loss $ave_loss out of $(size(X,1)) training images.")
    return W,b
end

end

"""
    predict_neural_network(X,W,b,num_hidden_layers,size_hidden_layers;activation,problem_type
,num_classes)
Predictions via Trained Neural Network
Computes predictions to input data X given weights W and biases b. Network settings should be
the same
as those which were used to learn W and b.
"""
function
predict_neural_network(X,W,b,num_hidden_layers,size_hidden_layers;activation="sigmoid",proble
m_type="classification",num_classes=1)
    # Create activation functions

```

```

if activation == "sigmoid"
     $\sigma = (z) \rightarrow 1 / (1 + \exp(-z))$ 
     $D\sigma = (z) \rightarrow \sigma(z)^2 * \exp(-z)$     # gradient with respect to z
    println("Sigmoid used.")
elseif activation == "ReLU"
     $\sigma = (z) \rightarrow \max(z, 0)$ 
     $D\sigma = (z) \rightarrow \begin{cases} 1.0 & \text{if } z \geq 0.0 \\ 0.0 & \text{else} \end{cases}$ 
    end
    println("ReLU used.")
end

# Create appropriate loss function
if problem_type == "classification"
    y_indicator = []
    classes = vcat(collect(1:9), 0)
    for i = 1:length(y)
        push!(y_indicator, map(k -> y[i] == k, classes))
    end

    L = (y_indicator, a_L_prob) -> -sum(y_indicator[k] * log(a_L_prob[k]) for
k=1:num_classes)
    DL = (y_indicator, a_L_prob) -> [-y_indicator[k] / a_L_prob[k] for k=1:num_classes]
    # These are gonna return vectors
    f = (z) -> [exp(z[j]) / sum(exp(z[k]) for k=1:num_classes) for j=1:num_classes]
    Df = (z) -> [(sum(exp(z[k]) for k=1:num_classes) - exp(z[j]))*exp(z[j]) /
(sum(exp(z[k]) for k=1:num_classes))^2 for j=1:num_classes]
elseif problem_type == "regression"
    L = (y, a_L) -> norm(y - a_L)^2
    DL = (y, a_L) -> 2*(a_L - y)
    f = (z) -> z
    Df = (z) -> 1
end

# Total number of layers
LL = num_hidden_layers + 2

# To store weighted inputs
z = map(_ -> zeros(size_hidden_layers), 1:num_hidden_layers)
pushfirst!(z, zeros(size(X,2)))
push!(z, zeros(num_classes))
# To store activations
a = map(_ -> zeros(size_hidden_layers), 1:num_hidden_layers)
pushfirst!(a, zeros(size(X,2)))
push!(a, zeros(num_classes))

```

```

# for l=2:LL
#     println("W_$l is:\n$(W[l])")
#     println("b_$l is:\n$(b[l])")
# end

function f_predictor(x)
    z[1] = x    # identical transformation
    a[1] = Vector{Float64}(σ.(z[1]))
    for l = 2:(LL-1)
        z[l] = W[l]*a[l-1] + b[l]
        a[l] = Vector{Float64}(σ.(z[l]))
    end
    z[LL] = W[LL]*a[LL-1] + b[LL]

    #println("$a[L]")

    if problem_type == "classification"
        a[LL] = f(z[LL])
        y_pred = argmax(a[LL])
        if y_pred == 10
            y_pred = 0
        end
        return y_pred
    elseif problem_type == "regression"
        a[LL] = f.(z[LL])
        return a[LL][1] # if regression, final layer has 1 output
    end
end

predictions = zeros(size(X,1))
for i = 1:size(X,1)
    predictions[i] = f_predictor(X[i,:])
end

return predictions
end

```