Assignment 1

Hongbo Du and 1003568089

February 24, 2020

The goal of this assignment is to get you familiar with the basics of decision theory and gradient-based model fitting.

1 Decision theory [13pts]

One successful use of probabilistic models is for building spam filters, which take in an email and take different actions depending on the likelihood that it's spam.

Imagine you are running an email service. You have a well-calibrated spam classifier that tells you the probability that a particular email is spam: $p(\text{spam}|\text{email})$. You have three options for what to do with each email: You can show it to the user, put it in the spam folder, or delete it entirely.

Depending on whether or not the email really is spam, the user will suffer a different amount of wasted time for the different actions we can take, $L(\text{action}, \text{spam})$:

Action	Spam	Not spam
Show	10	0
Folder	1	50
Delete	0	200

1. [3pts] Plot the expected wasted user time for each of the three possible actions, as a function of the probability of spam: $p(\text{spam}|\text{email})$

```julia
losses = [[10, 0],
          [1, 50],
          [0, 200]]

num_actions = length(losses)

function expected_loss_of_action(prob_spam, action)
    time = prob_spam * losses[action][1] + (1 .- prob_spam) * losses[action][2]
    return time
end

prob_range = range(0., stop=1., length=500)

# Make plot
using Plots
```

```
for action in 1:num_actions
  display(plot!(prob_range, expected_loss_of_action(prob_range, action),
   reuse = false))
end
```

2. [2pts] Write a function that computes the optimal action given the probability of spam.

```
function optimal_action(prob_spam)
    E_loss = []
    for action in 1:num_actions
        e_loss = expected_loss_of_action(prob_spam, action)
        E_loss = push!(E_loss, e_loss)
    end
    return findmin(E_loss)[2]
end
```

3. [4pts] Plot the expected loss of the optimal action as a function of the probability of spam.

Color the line according to the optimal action for that probability of spam.

```
function expected_loss_of_optimal_action(prob_spam)
    optimal_actions = []
    optimal_losses = []
    for p in prob_spam

        opt_act = optimal_action(p)
        optimal_actions = push!(optimal_actions, opt_act)

        opt_loss = expected_loss_of_action(p, opt_act)
        optimal_losses = push!(optimal_losses, opt_loss)

    end
    return optimal_actions, optimal_losses
end

plot(prob_range, expected_loss_of_optimal_action(prob_range)[2],
 linecolor=expected_loss_of_optimal_action(prob_range)[1],
 title = "Plot of the expected loss of the optimal action",
  label = "loss")
```

4. [4pts] For exactly which range of the probabilities of an email being spam should we delete an email?

Find the exact answer by hand using algebra.

Answer:

2 Regression

2.1 Manually Derived Linear Regression [10pts]

Suppose that $X \in \mathbb{R}^{m \times n}$ with $n \geq m$ and $Y \in \mathbb{R}^n$, and that $Y \sim \mathcal{N}(X^T\beta, \sigma^2 I)$.

In this question you will derive the result that the maximum likelihood estimate $\hat{\beta}$ of β is given by

$$\hat{\beta} = (XX^T)^{-1}XY$$

1. [1pts] What happens if $n < m$?
2. [2pts] What are the expectation and covariance matrix of $\hat{\beta}$, for a given true value of β?
3. [2pts] Show that maximizing the likelihood is equivalent to minimizing the squared error $\sum_{i=1}^{n}(y_i - x_i\beta)^2$. [Hint: Use $\sum_{i=1}^{n} a_i^2 = a^T a$]
4. [2pts] Write the squared error in vector notation, (see above hint), expand the expression, and collect like terms. [Hint: Use $\beta^T x^T y = y^T x \beta$ and $x^T x$ is symmetric]
5. [3pts] Use the likelihood expression to write the negative log-likelihood. Write the derivative of the negative log-likelihood with respect to β, set equal to zero, and solve to show the maximum likelihood estimate $\hat{\beta}$ as above.

2.2 Toy Data [2pts]

For visualization purposes and to minimize computational resources we will work with 1-dimensional toy data.

That is $X \in \mathbb{R}^{m \times n}$ where $m = 1$.

We will learn models for 3 target functions

- `target_f1`, linear trend with constant noise.
- `target_f2`, linear trend with heteroskedastic noise.
- `target_f3`, non-linear trend with heteroskedastic noise.

```julia
using LinearAlgebra

function target_f1(x, σ_true=0.3)
  noise = randn(size(x))
  y = 2x .+ σ_true.*noise
  return vec(y)
end

function target_f2(x)
  noise = randn(size(x))
  y = 2x + norm.(x)*0.3.*noise
  return vec(y)
end

function target_f3(x)
  noise = randn(size(x))
  y = 2x + 5sin.(0.5*x) + norm.(x)*0.3.*noise
  return vec(y)
end
```

1. [1pts] Write a function which produces a batch of data $x \sim \text{Uniform}(0, 20)$ and `y = target_f(x)`

```
using Random
using Distributions
Random.seed!(414)

function sample_batch(target_f, batch_size)
  x = transpose(rand(Uniform(0, 20), batch_size))
  y = target_f(x)
  return (x,y)
end

using Test
@testset "sample dimensions are correct" begin
  m = 1 # dimensionality
  n = 200 # batch-size
  for target_f in (target_f1, target_f2, target_f3)
    x,y = sample_batch(target_f,n)
    @test size(x) == (m,n)
    @test size(y) == (n,)
  end
end
```

2. [1pts] For all three targets, plot a $n = 1000$ sample of the data. **Note: You will use these plots later, in your writeup display once other questions are complete.**

```
using Plots

x1,y1 = sample_batch(target_f1,1000)
#plotplot(y1, seriestype=:scatter)
plot_f1 = scatter(transpose(x1), y1, title = "Plot of target_f1", label="target_f1")

x2,y2 = sample_batch(target_f2,1000)
plot_f2 = scatter(transpose(x2), y2, title = "Plot of target_f2", label="target_f2")

x3,y3 = sample_batch(target_f3,1000)
plot_f3 = scatter(transpose(x3), y3, title = "Plot of target_f3", label="target_f3")
```

2.3 Linear Regression Model with $\hat{\beta}$ MLE [4pts]

1. [2pts] Program the function that computes the the maximum likelihood estimate given X and Y. Use it to compute the estimate $\hat{\beta}$ for a $n = 1000$ sample from each target function.

```
function beta_mle(X,Y)
  beta = transpose(inv(X*transpose(X)))*X*Y
  return beta
end

n=1000 # batch_size

x_1, y_1 = sample_batch(target_f1, n)
β_mle_1 = beta_mle(x_1, y_1)
```

```
x_2, y_2 = sample_batch(target_f2, n)
β_mle_2 = beta_mle(x_2, y_2)

x_3, y_3 = sample_batch(target_f3, n)
β_mle_3 = beta_mle(x_3, y_3)
```

2. [2pts] For each function, plot the linear regression model given by $Y \sim \mathcal{N}(X^T\hat{\beta}, \sigma^2 I)$ for $\sigma = 1.$. This plot should have the line of best fit given by the maximum likelihood estimate, as well as a shaded region around the line corresponding to plus/minus one standard deviation (i.e. the fixed uncertainty $\sigma = 1.0$). Using `Plots.jl` this shaded uncertainty region can be achieved with the `ribbon` keyword argument. **Display 3 plots, one for each target function, showing samples of data and maximum likelihood estimate linear regression model**

```
σ = 1
plot!(plot_f1, [minimum(x_1), maximum(x_1)], [minimum(x_1), maximum(x_1)] .* β_mle_1;
 ribbon = (2σ), label = "best fit line")

σ = 1
plot!(plot_f2, [minimum(x_2), maximum(x_2)], [minimum(x_2), maximum(x_2)] .* β_mle_2;
 ribbon = (2σ), label = "best fit line")

σ = 1
plot!(plot_f3, [minimum(x_3), maximum(x_3)], [minimum(x_3), maximum(x_3)] .* β_mle_3;
 ribbon = (2σ), label = "best fit line")
```

2.4 Log-likelihood of Data Under Model [6pts]

1. [2pts] Write code for the function that computes the likelihood of x under the Gaussian distribution $\mathcal{N}(\mu, \sigma)$. For reasons that will be clear later, this function should be able to broadcast to the case where x, μ, σ are all vector valued and return a vector of likelihoods with equivalent length, i.e., $x_i \sim \mathcal{N}(\mu_i, \sigma_i)$.

```
function gaussian_log_likelihood(μ, σ, x)
  """
  compute log-likelihood of x under N(μ,σ)
  """
  log_likelihood = -length(x)/2*log.(2*π*σ.^2) - transpose(0.5*sum.(transpose(x .-
μ)*(x .- μ)/(σ.^2)))
  return log_likelihood
end

# Test Gaussian likelihood against standard implementation
@testset "Gaussian log likelihood" begin
using Distributions: pdf, Normal, logpdf
# Scalar mean and variance
x = randn()
μ = randn()
σ = rand()
@test size(gaussian_log_likelihood(μ,σ,x)) == () # Scalar log-likelihood
@test gaussian_log_likelihood.(μ,σ,x) ≈ log.(pdf.(Normal(μ,σ),x)) # Correct Value
# Vector valued x under constant mean and variance
```

```
x = randn(100)
μ = randn()
σ = rand()
@test size(gaussian_log_likelihood.(μ,σ,x)) == (100,) # Vector of log-likelihoods
@test gaussian_log_likelihood.(μ,σ,x) ≈ log.(pdf.(Normal(μ,σ),x)) # Correct Values
# Vector valued x under vector valued mean and variance
x = randn(10)
μ = randn(10)
σ = rand(10)
@test size(gaussian_log_likelihood.(μ,σ,x)) == (10,) # Vector of log-likelihoods
@test gaussian_log_likelihood.(μ,σ,x) ≈ logpdf.(Normal.(μ,σ),x) # Correct Values
end
```

2. [2pts] Use your gaussian log-likelihood function to write the code which computes the negative log-likelihood of the target value Y under the model $Y \sim \mathcal{N}(X^T\beta, \sigma^2 * I)$ for a given value of β.

```
function lr_model_nll(β,x,y, σ)
    res = gaussian_log_likelihood.(transpose(x)*β, σ^2, y)
    return sum((-1)*res)
end
```

3. [1pts] Use this function to compute and report the negative-log-likelihood of a $n \in \{10, 100, 1000\}$ batch of data under the model with the maximum-likelihood estimate $\hat{\beta}$ and $\sigma \in \{0.1, 0.3, 1., 2.\}$ for each target function.

```
for n in (10, 100, 1000) # batch_size
    println("--------  $n  ------------")
    for target_f in (target_f1,target_f2, target_f3) # target_f
      println("--------  $target_f  ------------")
      for σ_model in (0.1,0.3,1.,2.)
        println("--------  $σ_model  ------------")
        x,y = sample_batch(target_f, n)
        β_mle = beta_mle(x,y)
        nll = lr_model_nll(β_mle,x,y, σ_model)
        println("Negative Log-Likelihood: $nll")
      end
    end
end
```

4. [1pts] For each target function, what is the best choice of σ?

Please note that σ and batch-size n are modelling hyperparameters. In the expression of maximum likelihood estimate, σ or n do not appear, and in principle shouldn't affect the final answer. However, in practice these can have significant effect on the numerical stability of the model. Too small values of σ will make data away from the mean very unlikely, which can cause issues with precision. Also, the negative log-likelihood objective involves a sum over the log-likelihoods of each datapoint. This means that with a larger batch-size n, there are more datapoints to sum over, so a larger negative log-likelihood is not necessarily worse. The take-home is that you cannot directly compare the negative log-likelihoods achieved by these models with different hyperparameter settings.

2.5 Automatic Differentiation and Maximizing Likelihood [3pts]

In a previous question you derived the expression for the derivative of the negative log-likelihood with respect to β. We will use that to test the gradients produced by automatic differentiation.

1. [3pts] For a random value of β, σ, and $n = 100$ sample from a target function, use automatic differentiation to compute the derivative of the negative log-likelihood of the sampled data with respect β. Test that this is equivalent to the hand-derived value.

```
using Zygote: gradient

@testset "Gradients wrt parameter" begin
β_test = randn()
σ_test = rand()
x,y = sample_batch(target_f1,100)
ad_grad = gradient(β_test -> lr_model_nll(β_test,x,y, σ_test), β_test)
hand_derivative = sum(length(x)/(2σ.^2)* (x*transpose(x)*β_test - x*y))
@test isapprox(ad_grad[1], hand_derivative, atol=1e7)
end
```

2.5.1 Train Linear Regression Model with Gradient Descent [5pts]

In this question we will compute gradients of of negative log-likelihood with respect to β. We will use gradient descent to find β that maximizes the likelihood.

1. [3pts] Write a function `train_lin_reg` that accepts a target function and an initial estimate for β and some hyperparameters for batch-size, model variance, learning rate, and number of iterations. Then, for each iteration:

   – sample data from the target function

   – compute gradients of negative log-likelihood with respect to β

   – update the estimate of β with gradient descent with specified learning rate

   and, after all iterations, returns the final estimate of β.

```
using Logging # Print training progress to REPL, not pdf

function train_lin_reg(target_f, β_init; bs= 100, lr = 1e-6, iters=1000, σ_model = 1. )
    β_curr = β_init
    for i in 1:iters
      x,y = sample_batch(target_f, bs)
      #@info "loss:$()  β: $β_curr"
      grad_β = gradient(β_curr -> lr_model_nll(β_curr,x,y, σ_model), β_curr)
      β_prev = β_curr
      β_curr = β_curr - lr * grad_β[1]
    end
    return β_curr
end
```

2. [2pts] For each target function, start with an initial parameter β, learn an estimate for β_{learned} by gradient descent. Then plot a $n = 1000$ sample of the data and the learned linear regression model with shaded region for uncertainty corresponding to plus/minus one standard deviation.

```
β_init = 1000*randn() # Initial parameter
β_learned1 = train_lin_reg(target_f1, β_init; bs= 100, lr = 1e-6, iters=1000, σ_model =
1. )
β_learned2 = train_lin_reg(target_f2, β_init; bs= 100, lr = 1e-6, iters=1000, σ_model =
1. )
β_learned3 = train_lin_reg(target_f3, β_init; bs= 100, lr = 1e-6, iters=1000, σ_model =
1. )

σ = 1
plot!(plot_f1, [minimum(x_1), maximum(x_1)], [minimum(x_1), maximum(x_1)] .* β_learned1;
 ribbon = (2σ), label = "beta_learned1", title = "Plot for beta_learned1")

plot!(plot_f2, [minimum(x_1), maximum(x_1)], [minimum(x_1), maximum(x_1)] .* β_learned2;
 ribbon = (2σ), label = "beta_learned2", title = "Plot for beta_learned2")

plot!(plot_f3, [minimum(x_1), maximum(x_1)], [minimum(x_1), maximum(x_1)] .* β_learned3;
 ribbon = (2σ), label = "beta_learned3", title = "Plot for beta_learned3")
```

2.5.2 Non-linear Regression with a Neural Network [9pts]

In the previous questions we have considered a linear regression model

$$Y \sim \mathcal{N}(X^T\beta, \sigma^2)$$

This model specified the mean of the predictive distribution for each datapoint by the product of that datapoint with our parameter.

Now, let us generalize this to consider a model where the mean of the predictive distribution is a non-linear function of each datapoint. We will have our non-linear model be a simple function called `neural_net` with parameters θ (collection of weights and biases).

$$Y \sim \mathcal{N}(\texttt{neural\_net}(X, \theta), \sigma^2)$$

1. [3pts] Write the code for a fully-connected neural network (multi-layer perceptron) with one 10-dimensional hidden layer and a `tanh` nonlinearirty. You must write this yourself using only basic operations like matrix multiply and `tanh`, you may not use layers provided by a library.

   This network will output the mean vector, test that it outputs the correct shape for some random parameters.

```
# Neural Network Function
function neural_net(x,θ)
    hidden_layer = tanh.(x' * θ[1] .+ θ[2])
    output_layer = hidden_layer * θ[3] .+ θ[4]
  return output_layer
end
```

```
# Random initial Parameters
θ = (rand(1, 10), rand(1, 10), rand(10, 1), rand(1, 1))

@testset "neural net mean vector output" begin
n = 100
x,y = sample_batch(target_f1,n)
μ = neural_net(x,θ)
@test size(μ) == (n, 1)
end
```

2. [2pts] Write the code that computes the negative log-likelihood for this model where the mean is given by the output of the neural network and $\sigma = 1.0$

```
function nn_model_nll(θ,x,y;σ=1)
  return sum((-1)*gaussian_log_likelihood.(neural_net(x,θ), σ, y))
end
```

3. [2pts] Write a function `train_nn_reg` that accepts a target function and an initial estimate for θ and some hyperparameters for batch-size, model variance, learning rate, and number of iterations. Then, for each iteration:

   – sample data from the target function

   – compute gradients of negative log-likelihood with respect to θ

   – update the estimate of θ with gradient descent with specified learning rate

   and, after all iterations, returns the final estimate of θ.

```
using Logging # Print training progress to REPL, not pdf

function train_nn_reg(target_f, θ_init; bs= 100, lr = 1e-5, iters=1000, σ_model = 1. )
    θ_curr = θ_init
    for i in 1:iters
      x,y = sample_batch(target_f, bs)
      #@info "loss: $()" #TODO: log loss, if you want to montior training
      grad_θ = gradient(θ_curr -> nn_model_nll(θ_curr,x,y;σ=σ_model), θ_curr)
      θ_curr = θ_curr .- lr .* grad_θ[1]
    end
    return θ_curr
end
```

4. [2pts] For each target function, start with an initialization of the network parameters, θ, use your train function to minimize the negative log-likelihood and find an estimate for θ_{learned} by gradient descent. Then plot a $n = 1000$ sample of the data and the learned regression model with shaded uncertainty bounds given by $\sigma = 1.0$

```
θ_init = (rand(1, 10), rand(1, 10), rand(10, 1), rand(1, 1))
θ_learned_1 = train_nn_reg(target_f1, θ_init; bs= 1000, lr = 1e-5, iters=1000, σ_model =
1. )
θ_learned_2 = train_nn_reg(target_f2, θ_init; bs= 1000, lr = 1e-5, iters=1000, σ_model =
1. )
θ_learned_3 = train_nn_reg(target_f3, θ_init; bs= 1000, lr = 1e-5, iters=1000, σ_model =
1. )
```

```
σ = 1
y_hat1 = neural_net(x_1, θ_learned_1)
plot!(plot_f1, vec(x_1), vec(y_hat1);
 ribbon = (2σ), label = "theta_learned1", title = "Plot for theta_learned1")

y_hat2 = neural_net(x_2, θ_learned_2)
plot!(plot_f2, vec(x_2), vec(y_hat2);
 ribbon = (2σ), label = "theta_learned2", title = "Plot for theta_learned2")

y_hat3 = neural_net(x_3, θ_learned_3)
plot!(plot_f3, vec(x_3), vec(y_hat3);
 ribbon = (2σ), label = "theta_learned3", title = "Plot for theta_learned3")
```

2.5.3 Non-linear Regression and Input-dependent Variance with a Neural Network [8pts]

In the previous questions we've gone from a gaussian model with mean given by linear combination

$$Y \sim \mathcal{N}(X^T\beta, \sigma^2)$$

to gaussian model with mean given by non-linear function of the data (neural network)

$$Y \sim \mathcal{N}(\texttt{neural\_net}(X, \theta), \sigma^2)$$

However, in all cases we have considered so far, we specify a fixed variance for our model distribution. We know that two of our target datasets have heteroscedastic noise, meaning any fixed choice of variance will poorly model the data.

In this question we will use a neural network to learn both the mean and log-variance of our gaussian model.

$$\mu, \log\sigma = \texttt{neural\_net}(X, \theta)$$
$$Y \sim \mathcal{N}(\mu, \exp(\log\sigma)^2)$$

1. [1pts] Write the code for a fully-connected neural network (multi-layer perceptron) with one 10-dimensional hidden layer and a `tanh` nonlinearirty, and outputs both a vector for mean and $\log\sigma$. Test the output shape is as expected.

```
# Neural Network Function
function neural_net_w_var(x,θ)
  hidden_layer = tanh.(x' * θ[1] .+ θ[2])
  output_layer = hidden_layer * θ[3] .+ θ[4]
  return (output_layer, output_layer)
end

# Random initial Parameters
θ = (rand(1, 10), rand(1, 10), rand(10, 1), rand(1, 1))

@testset "neural net mean and logsigma vector output" begin
n = 100
```

```
x,y = sample_batch(target_f1,n)
μ, logσ = neural_net_w_var(x,θ)
@test size(μ) == (n,1)
@test size(logσ) == (n,1)
end
```

2. [2pts] Write the code that computes the negative log-likelihood for this model where the mean and $\log\sigma$ is given by the output of the neural network. (Hint: Don't forget to take $\exp\log\sigma$)

```
function nn_with_var_model_nll(θ,x,y)
  return sum((-1)*gaussian_log_likelihood.(neural_net_w_var(x,θ)[1],
exp.(neural_net_w_var(x,θ)[2]).^2, y))
end
```

3. [1pts] Write a function `train_nn_w_var_reg` that accepts a target function and an initial estimate for θ and some hyperparameters for batch-size, learning rate, and number of iterations. Then, for each iteration:

   – sample data from the target function

   – compute gradients of negative log-likelihood with respect to θ

   – update the estimate of θ with gradient descent with specified learning rate

   and, after all iterations, returns the final estimate of θ.

```
function train_nn_w_var_reg(target_f, θ_init; bs= 100, lr = 1e-4, iters=10000)
    θ_curr = θ_init
    for i in 1:iters
      x,y = sample_batch(target_f, bs)
      #@info "loss: $()" #TODO: log loss
      grad_θ = gradient(θ_curr -> nn_with_var_model_nll(θ_curr,x,y), θ_curr)
      θ_curr = θ_curr .- lr .* grad_θ[1]
    end
    return θ_curr
end
```

4. [4pts] For each target function, start with an initialization of the network parameters, θ, learn an estimate for θ_{learned} by gradient descent. Then plot a $n = 1000$ sample of the dataset and the learned regression model with shaded uncertainty bounds corresponding to plus/minus one standard deviation given by the variance of the predictive distribution at each input location (output by the neural network). (Hint: `ribbon` argument for shaded uncertainty bounds can accept a vector of σ)

   Note: Learning the variance is tricky, and this may be unstable during training. There are some things you can try:

   – Adjusting the hyperparameters like learning rate and batch size

   – Train for more iterations

   – Try a different random initialization, like sample random weights and bias matrices with lower variance.

For this question **you will not be assessed on the final quality of your model**. Specifically, if you fails to train an optimal model for the data that is okay. You are expected to learn something that is somewhat reasonable, and **demonstrates that this model is training and learning variance**.

If your implementation is correct, it is possible to learn a reasonable model with fewer than 10 minutes of training on a laptop CPU. The default hyperparameters should help, but may need some tuning.

```
θ_init = (rand(1, 10), rand(1, 10), rand(10, 1), rand(1, 1))
θ_learned1 = train_nn_w_var_reg(target_f1, θ_init; bs= 1000, lr = 1e-4, iters=1000)
θ_learned2 = train_nn_w_var_reg(target_f2, θ_init; bs= 1000, lr = 1e-4, iters=1000)
θ_learned3 = train_nn_w_var_reg(target_f3, θ_init; bs= 1000, lr = 1e-4, iters=1000)

mu_var_hat1 = neural_net_w_var(x_1, θ_learned1)[1]
logσ1 = exp.(neural_net_w_var(x_1, θ_learned1)[2]).^2
plot!(plot_f1, vec(x_1), vec(mu_var_hat1);
 ribbon = (2logσ1), label = "theta_var_learned1", title = "Plot for theta_var_learned1")

mu_var_hat2 = neural_net_w_var(x_2, θ_learned2)[1]
logσ2 = exp.(neural_net_w_var(x_2, θ_learned2)[2]).^2
plot!(plot_f2, vec(x_2), vec(mu_var_hat2);
 ribbon = (2logσ2), label = "theta_var_learned2", title = "Plot for theta_var_learned2")

mu_var_hat3 = neural_net_w_var(x_3, θ_learned3)[1]
logσ3 = exp.(neural_net_w_var(x_3, θ_learned3)[2]).^2
plot!(plot_f3, vec(x_3), vec(mu_var_hat3);
 ribbon = (2logσ3), label = "theta_var_learned3", title = "Plot for theta_var_learned3")


#mu_var_hat1 = neural_net_w_var(x_2, θ_learned2)[1]
#logσ2 = exp.(neural_net_w_var(x_2, θ_learned2)[2]).^2
#plot!(plot_f2, [minimum(x_2), maximum(x_2)], [minimum(logσ2), maximum(logσ2)];
# ribbon = (2logσ2), label = "theta_var_learned2", title = "Plot for
theta_var_learned2")

#mu_var_hat3 = neural_net_w_var(x_3, θ_learned3)[1]
#logσ3 = exp.(neural_net_w_var(x_3, θ_learned3)[2]).^2
#plot!(plot_f1, [minimum(x_3), maximum(x_3)], [minimum(logσ3), maximum(logσ3)];
# ribbon = (2logσ3), label = "theta_var_learned3", title = "Plot for
theta_var_learned3")
```

If you would like to take the time to train a very good model of the data (specifically for target functions 2 and 3) with a neural network that outputs both mean and $\log \sigma$ you can do this, but it is not necessary to achieve full marks. You can try

- Using a more stable optimizer, like Adam. You may import this from a library.
- Increasing the expressivity of the neural network, increase the number of layers or the dimensionality of the hidden layer.
- Careful tuning of hyperparameters, like learning rate and batchsize.