

Neural Networks

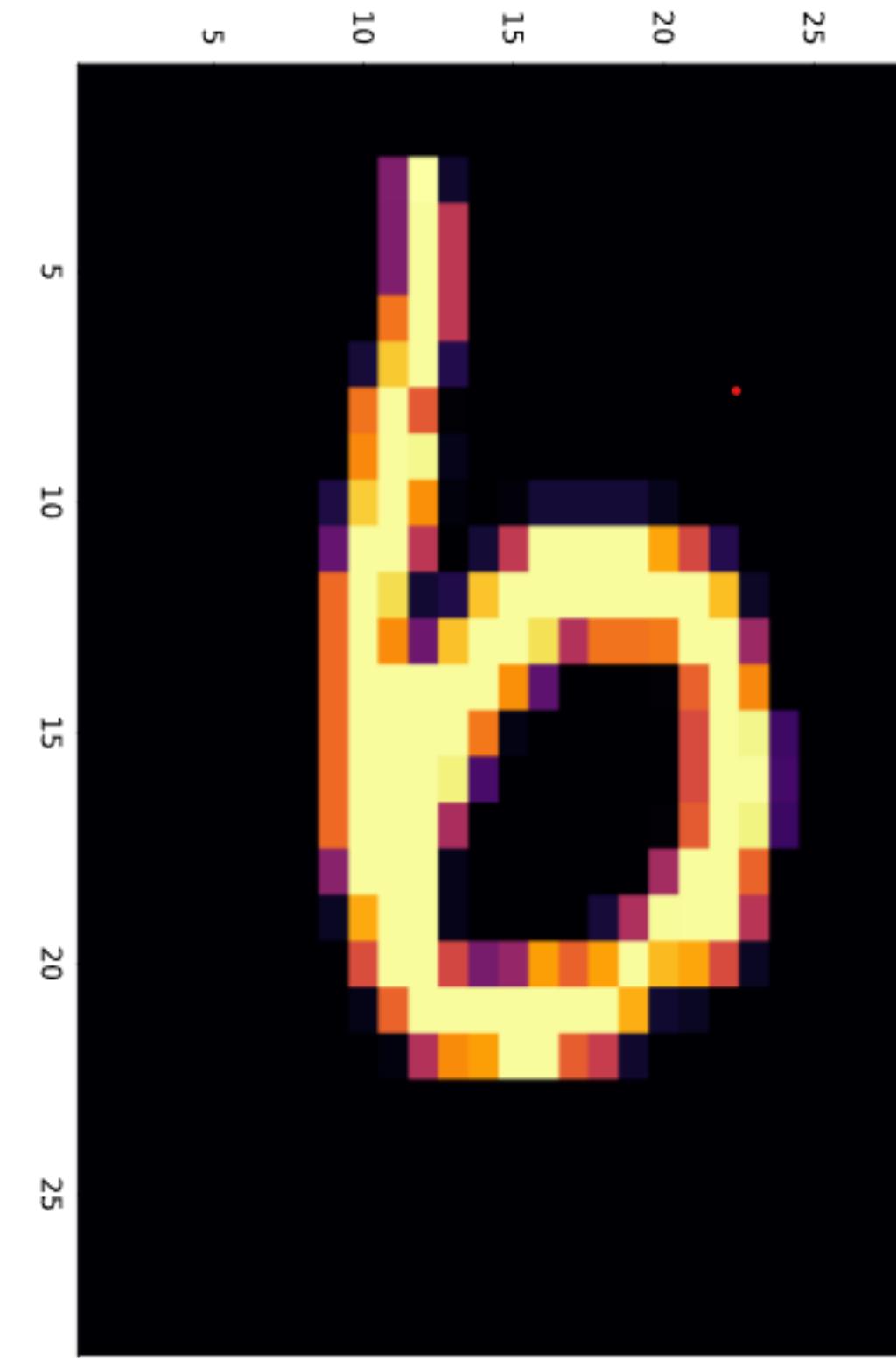
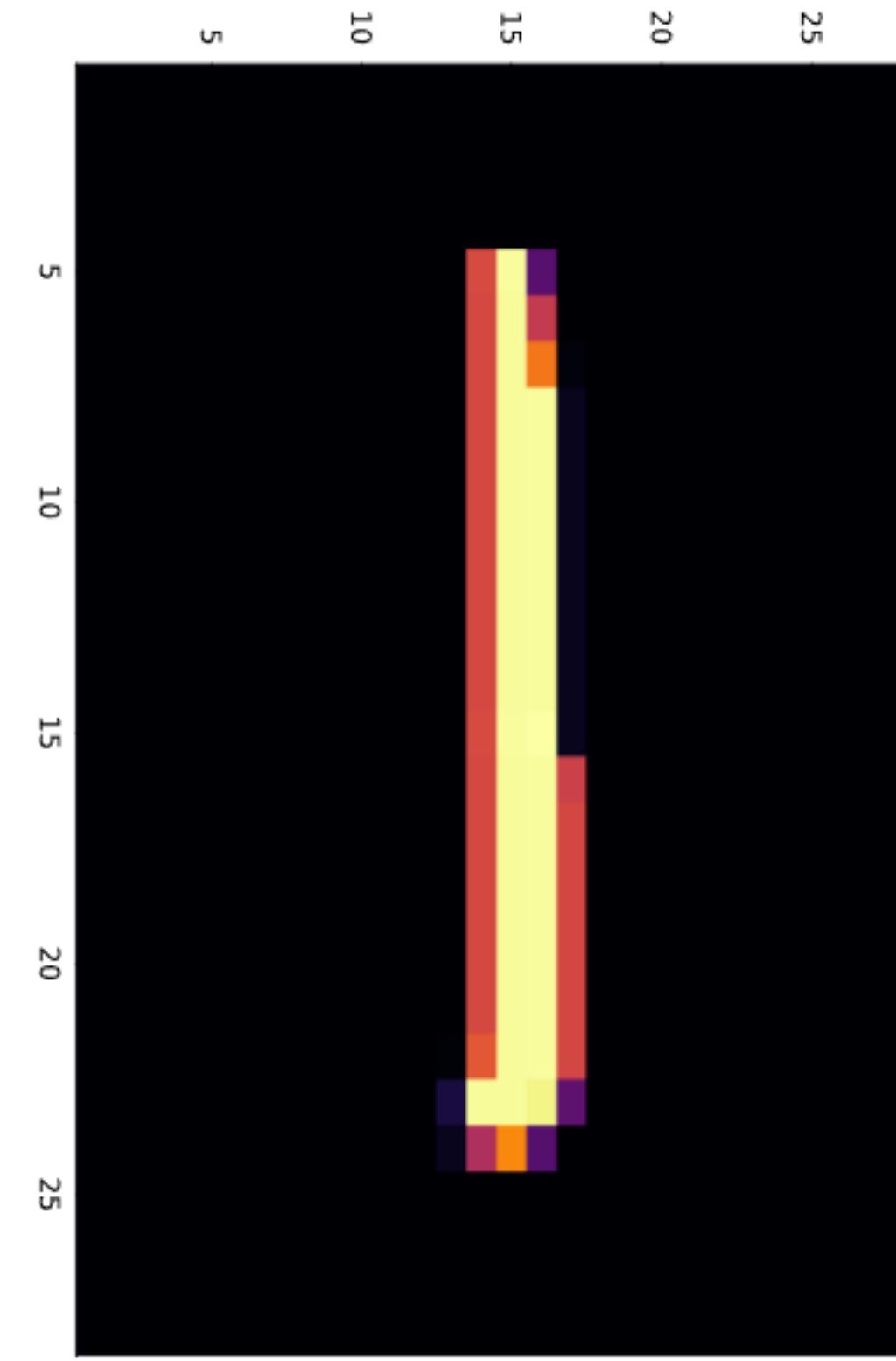
A brief introduction to Neural Networks

Felipe Whitaker, 14/04/2023

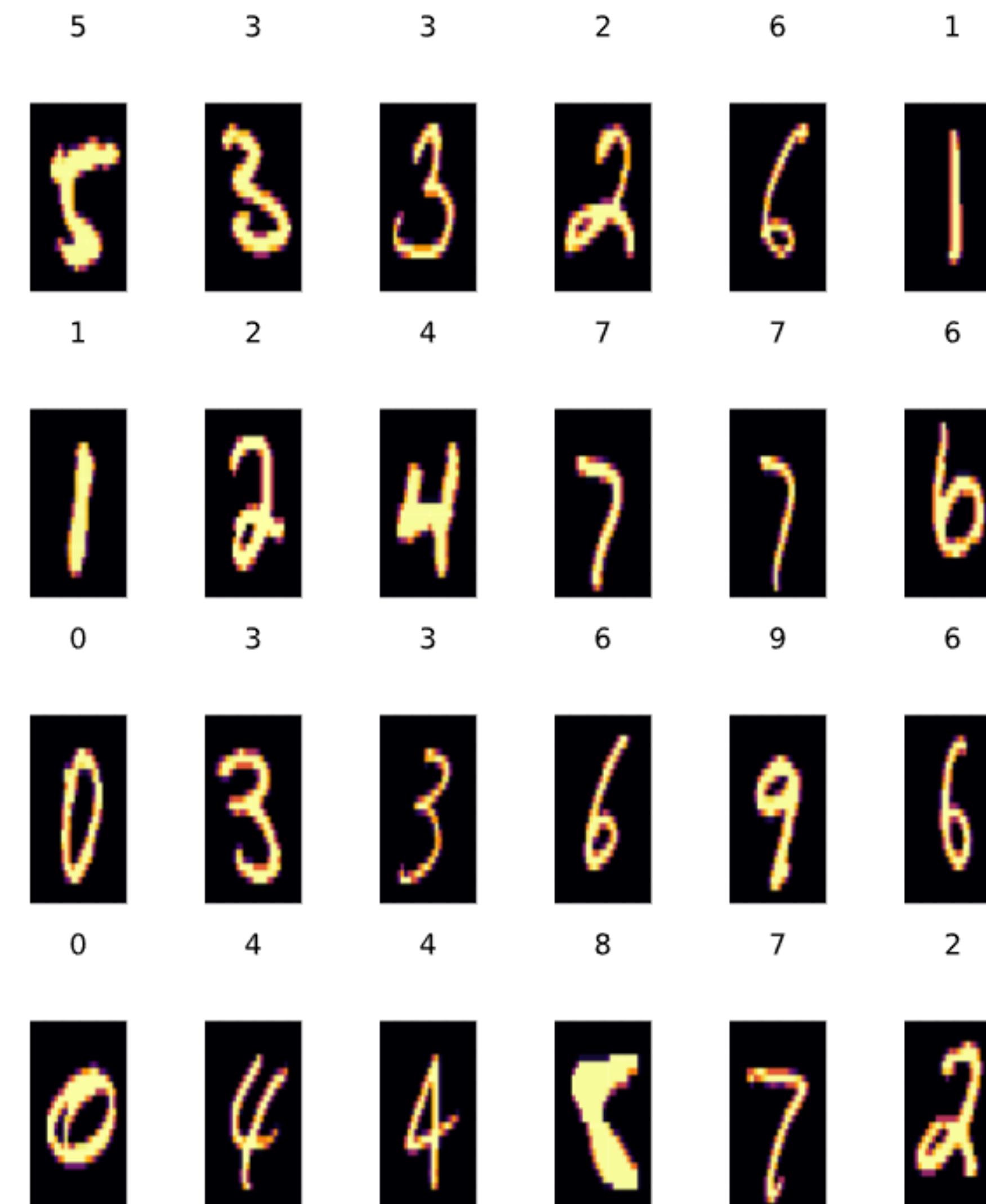
Motivation

What is signal?
& How to extract it?

Hand Writing Recognition



Hand Writing Recognition



Hand Writing Recognition

5x2



3x7



3x2



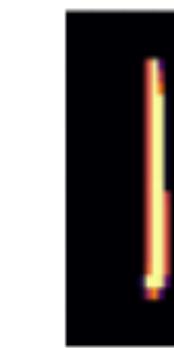
2x2



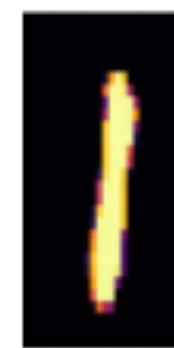
6x2



1x5



1x2



2x7



4x2



7x2



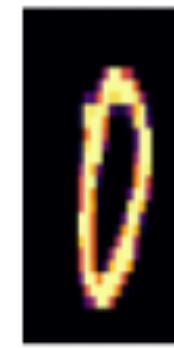
7x2



6x2



0x2



3x2



3x2



6x2



9x2



6x2



0x2



4x2



4x2



8x7



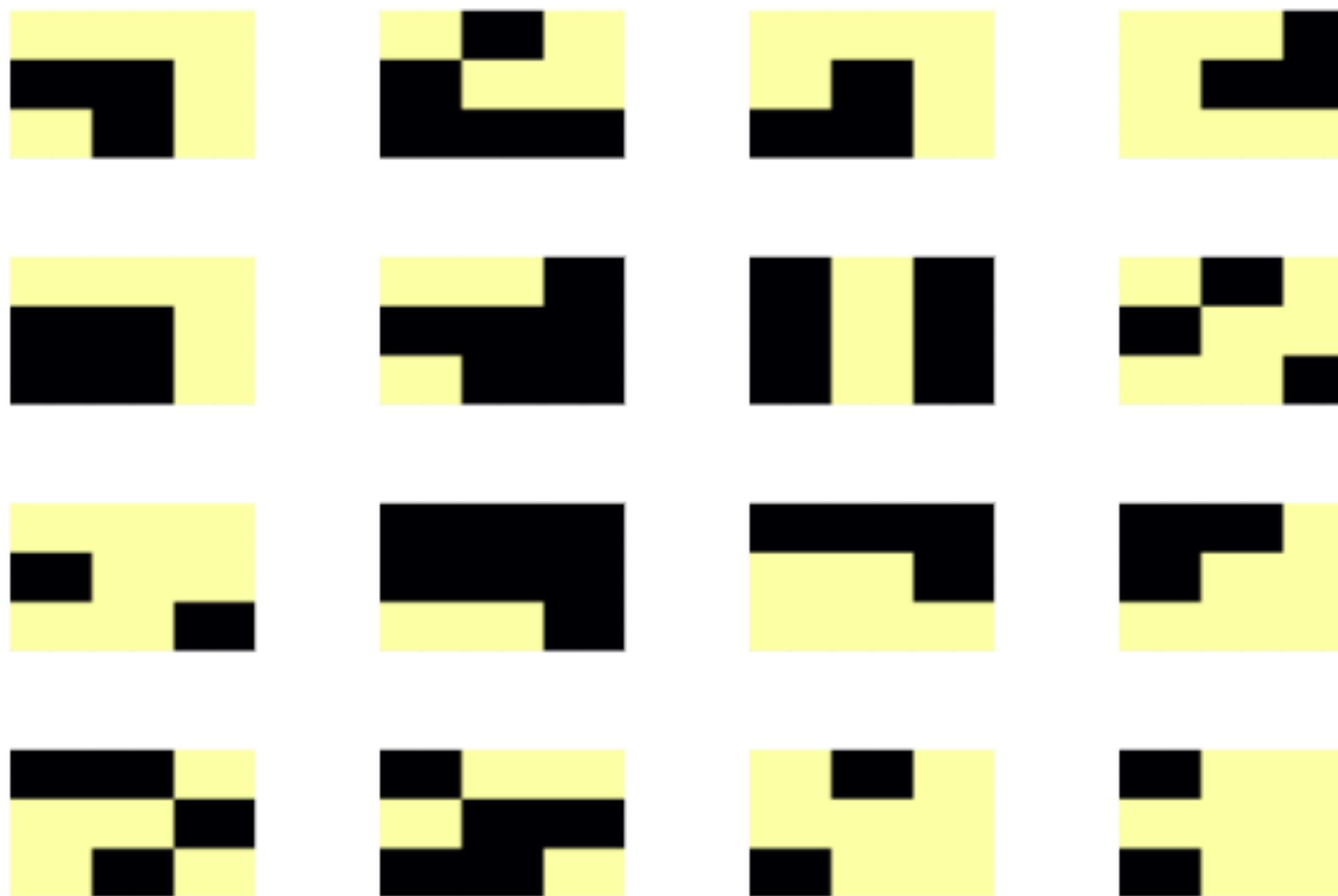
7x2



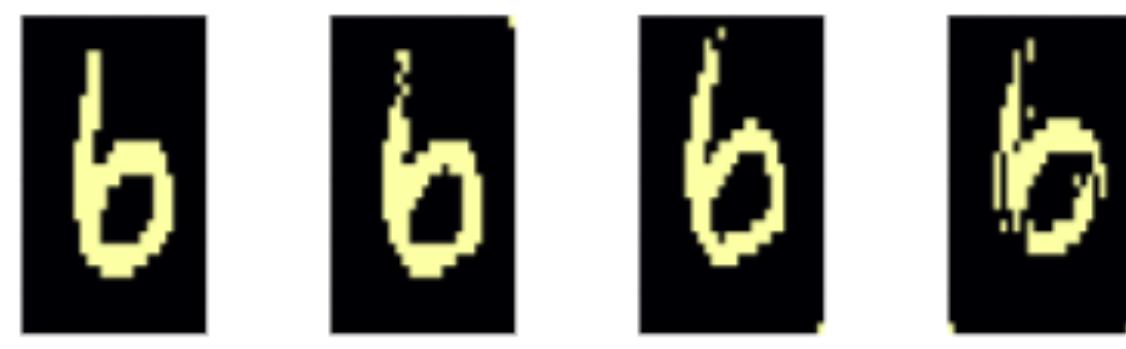
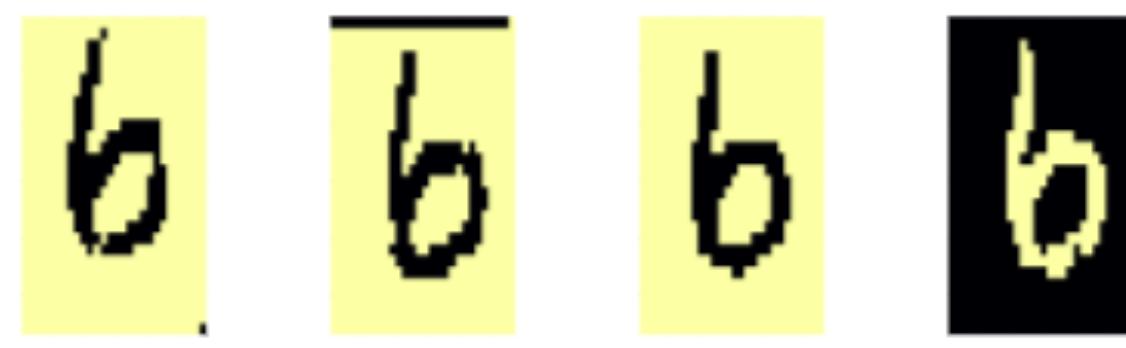
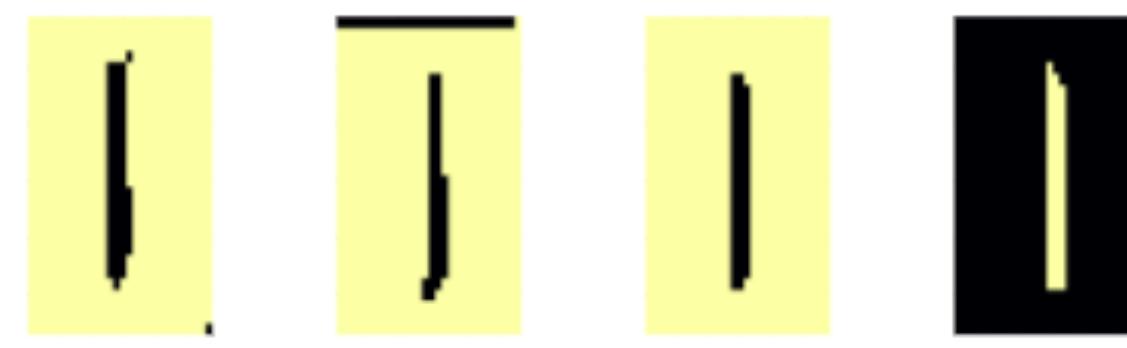
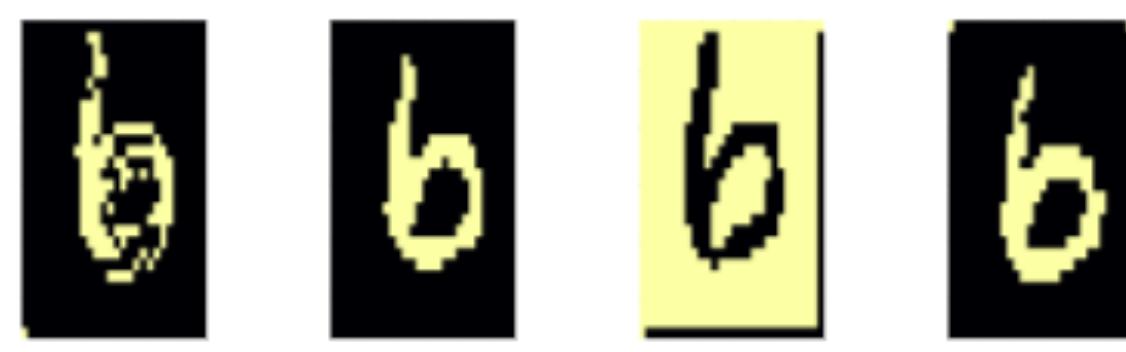
2x6



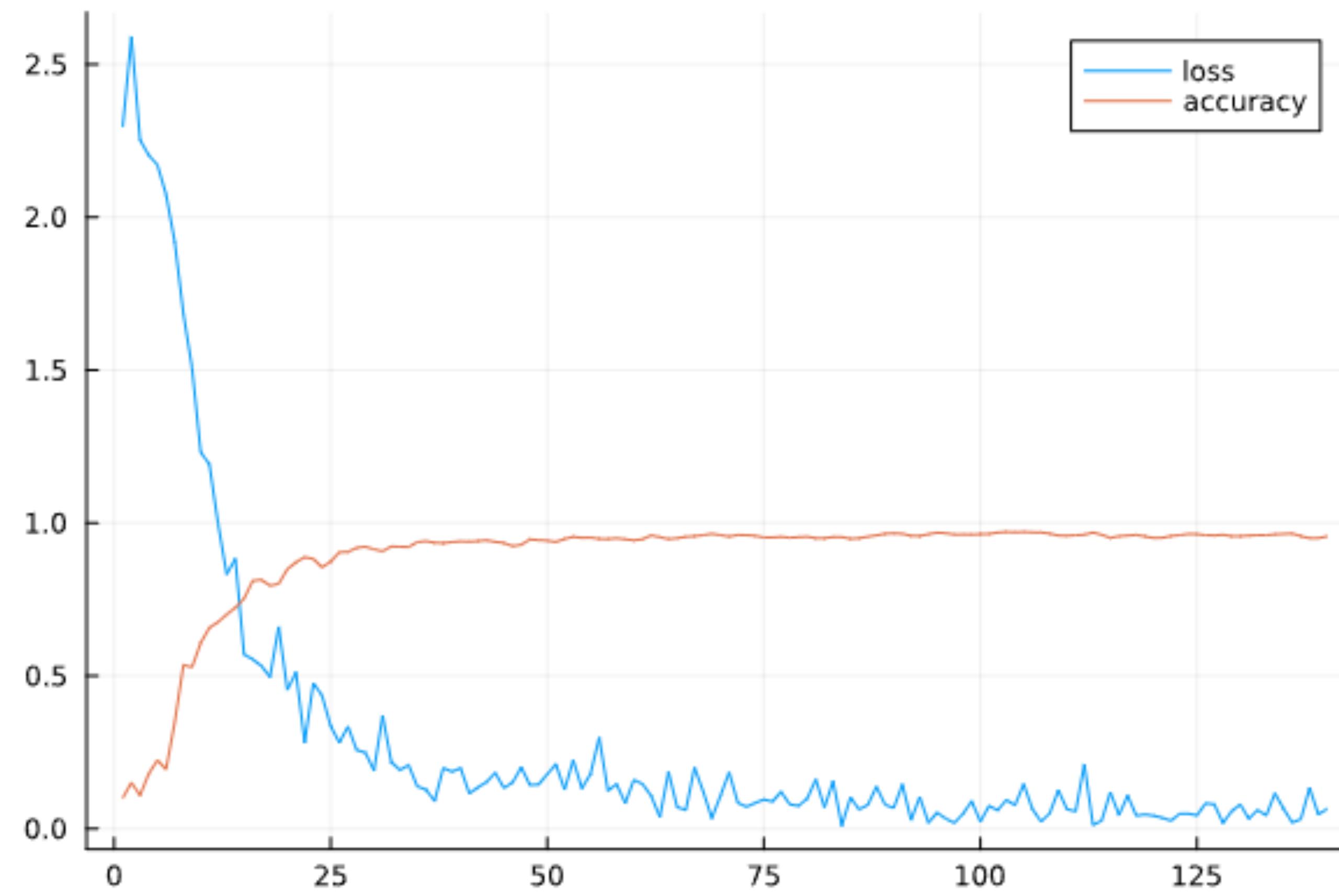
Hand Writing Recognition



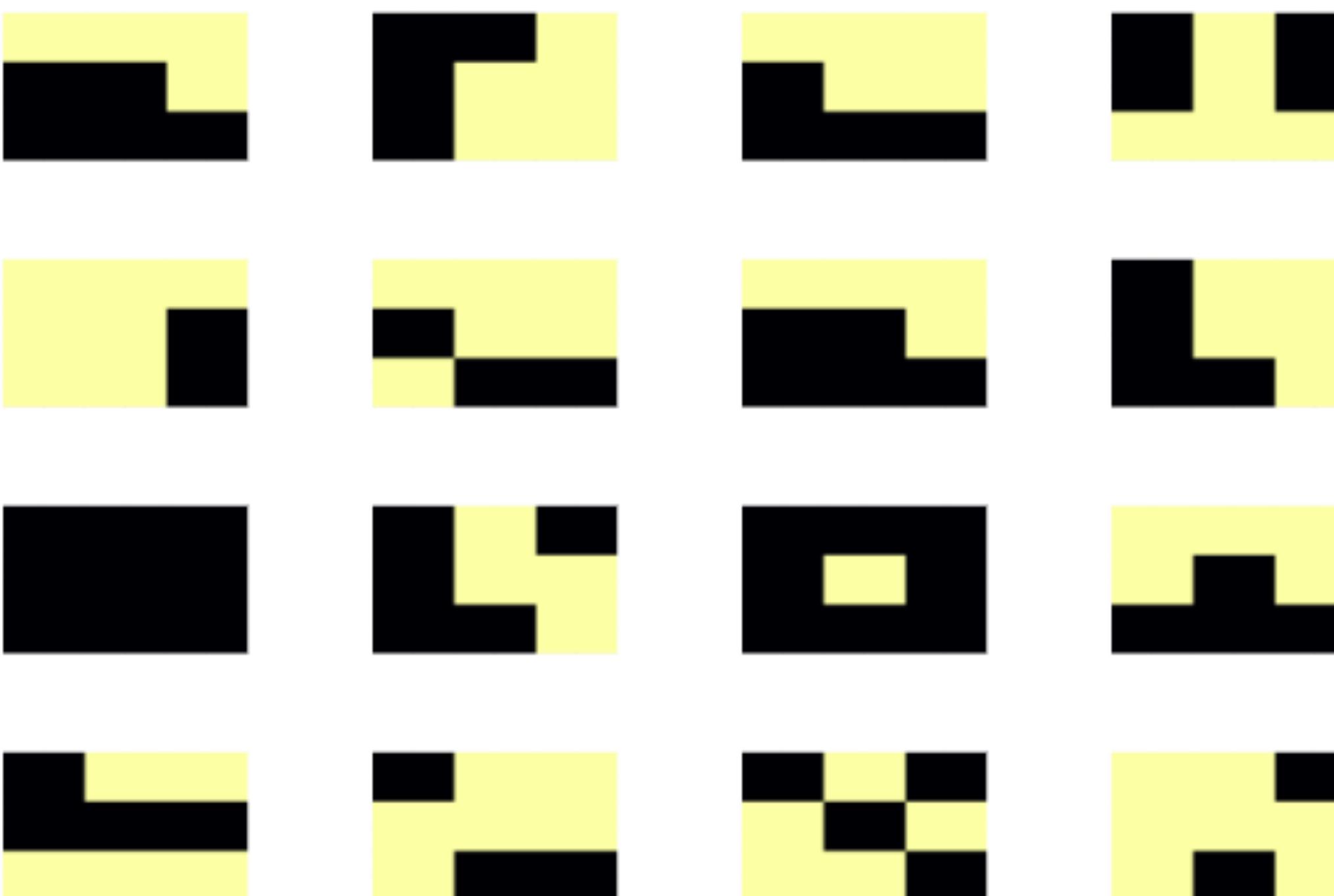
Hand Writing Recognition



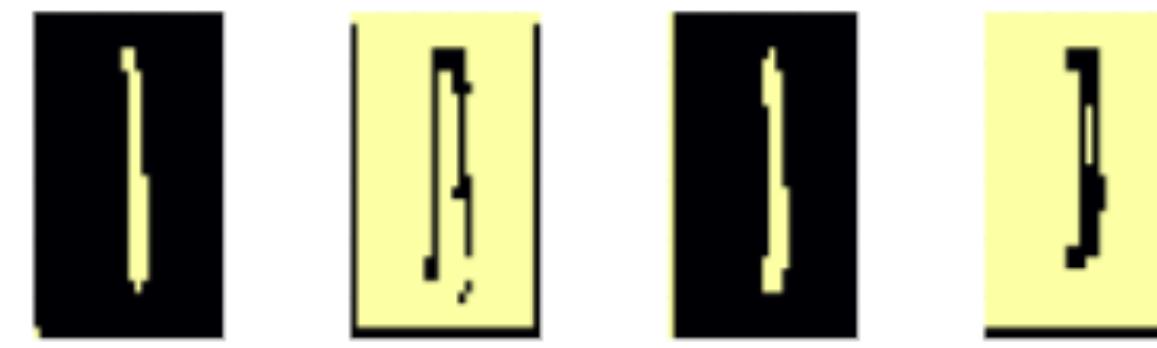
Hand Writing Recognition



Hand Writing Recognition



Hand Writing Recognition



Hand Writing Recognition

5x5



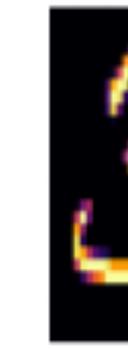
1x1

3x2



2x2

3x3



4x4

2x2



7x7

6x6



7x7

1x1



6x6

1

0x0

2

3x3

4

3x3

7

6x6

7

9x9

6

6x6

0

0x0

3

4x4

3

4x4

6

8x8

9

7x7

6

2x2

0

11

4

4

5

7

2

Summary

- Linear Regression
 - Closed Form
 - **Gradient Descent**
- Feature Engineering
- Feed Forward Model
 - **Learning a non-linear function**
 - Classification
- Convolutional Neural Networks
- Recurrent Neural Networks

Linear Regression

$$y = f(x) = \beta_0 + \beta_1 \cdot x + \epsilon$$

$$\frac{\partial J}{\partial \beta} = 0 \rightarrow \text{Closed Form}$$

./code/lm-vector.jl

$$J(\beta) = \frac{1}{2} \sum_{i=1}^N (y - \hat{y})^2 = \frac{1}{2} \sum_{i=1}^N (y - X \cdot \beta)^2$$

$$\frac{\partial J}{\partial \beta} = X^T \cdot (\beta \cdot X - y) \rightarrow \text{Gradient Descent}$$

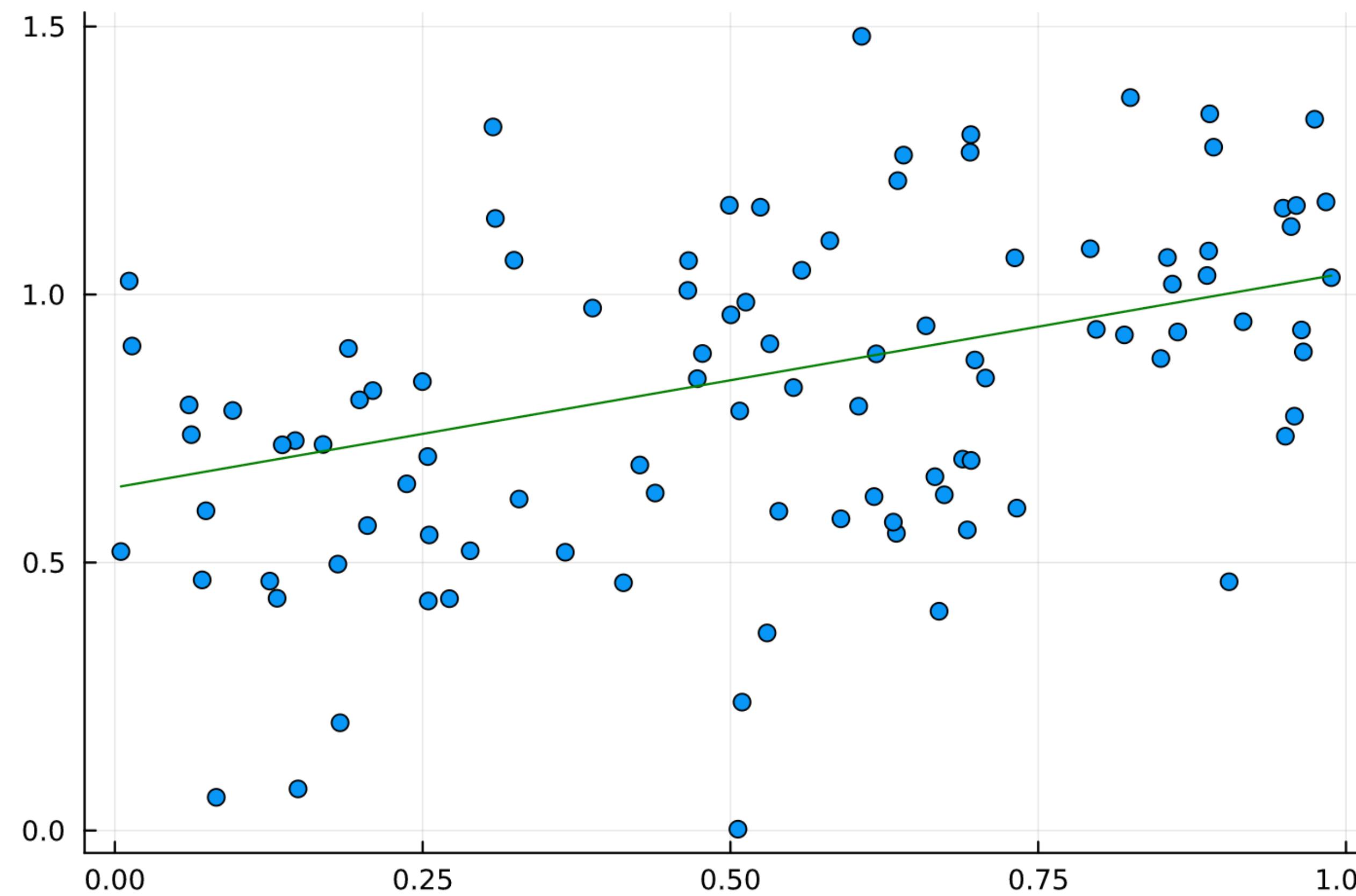
./code/lm-grad.jl

Linear Regression

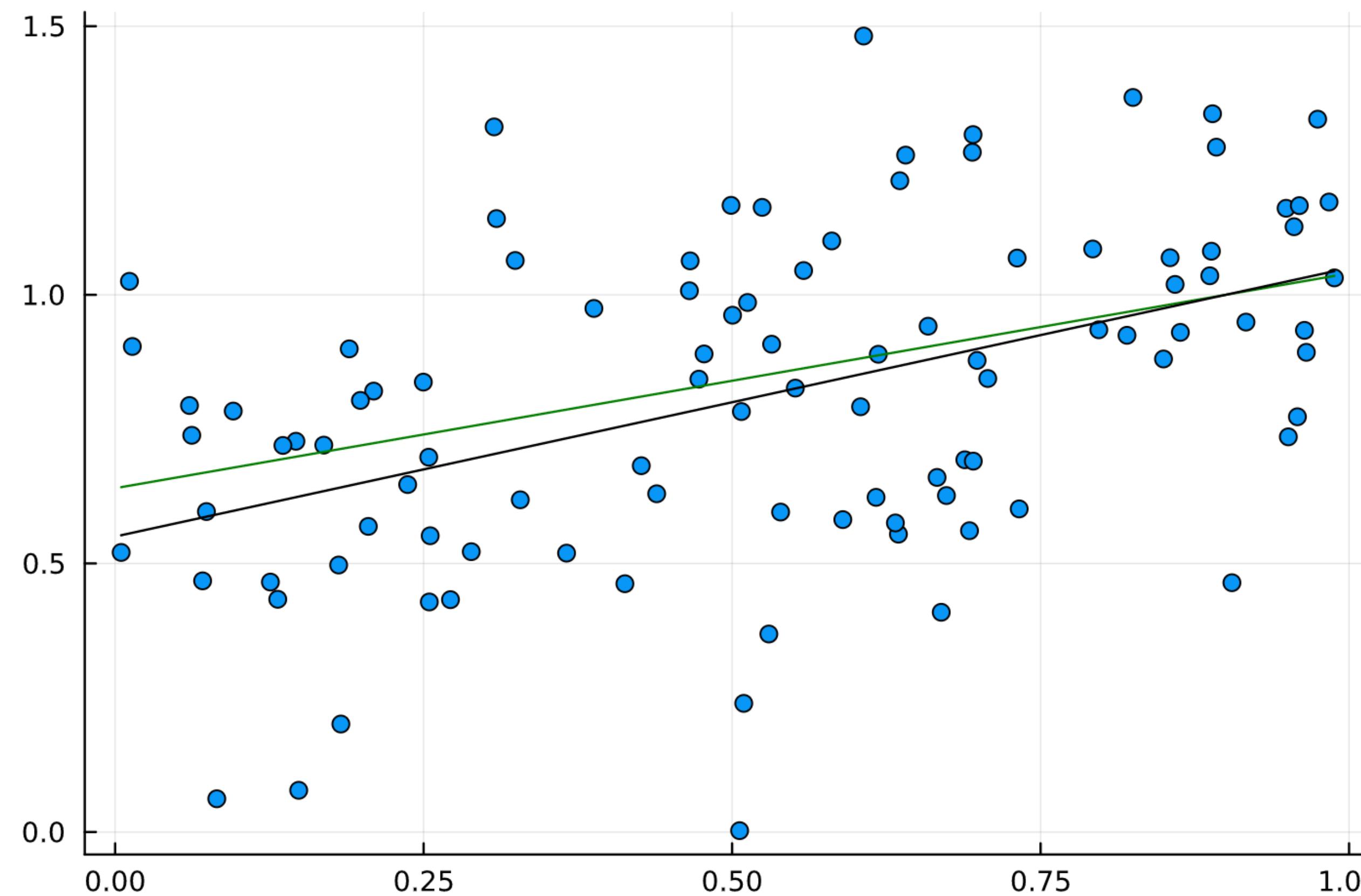
$$y = f(x) = \beta_0 + \beta_1 \cdot x + \epsilon$$

- Closed Form: $\beta = (X^T \cdot X)^{-1} \cdot X^T \cdot Y$
- Gradient Descent: $\beta - \eta \cdot \frac{\partial J}{\partial \beta} = \beta - \eta \cdot X^T \cdot (X \cdot \beta - y)$

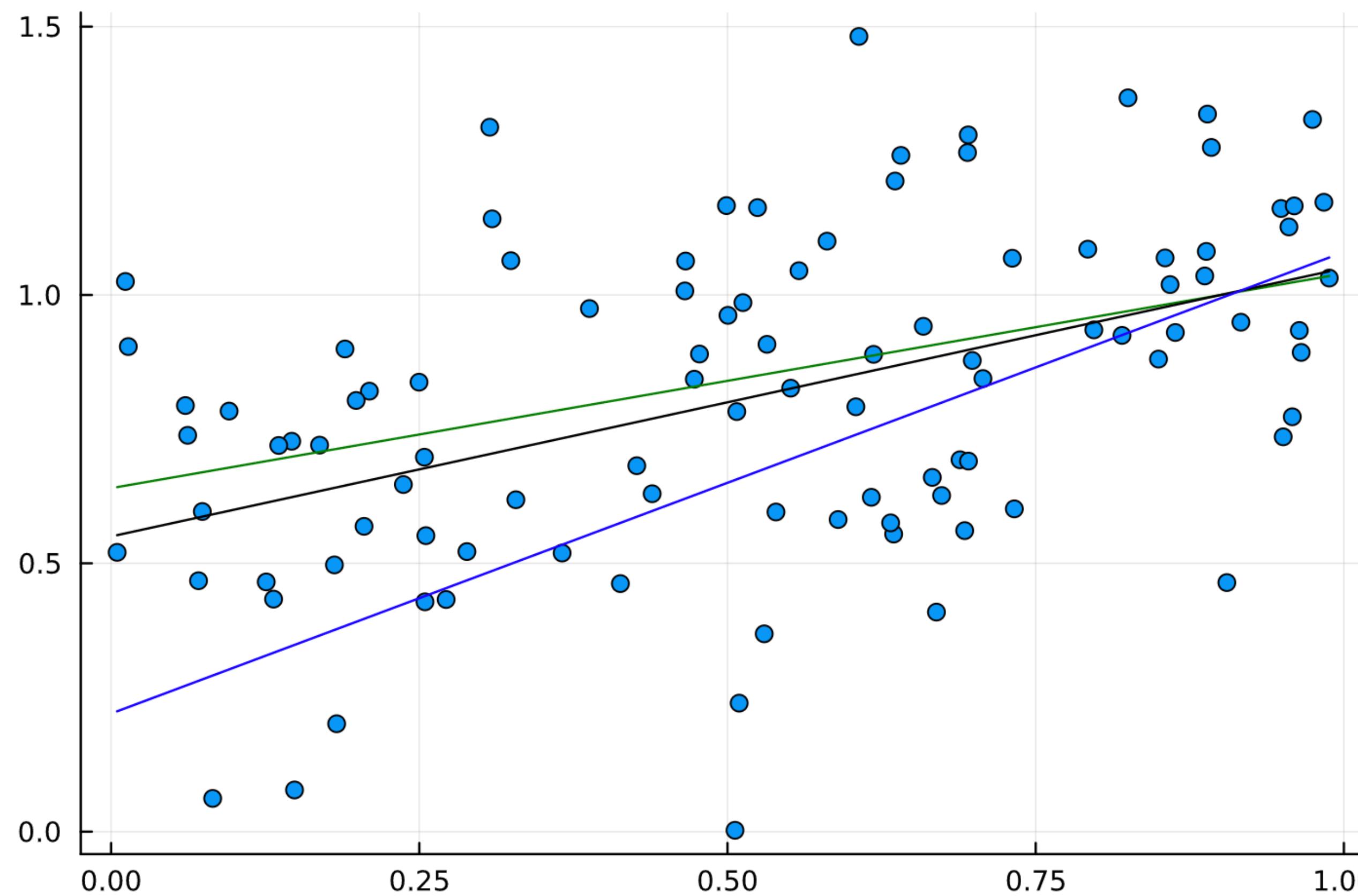
Linear Regression



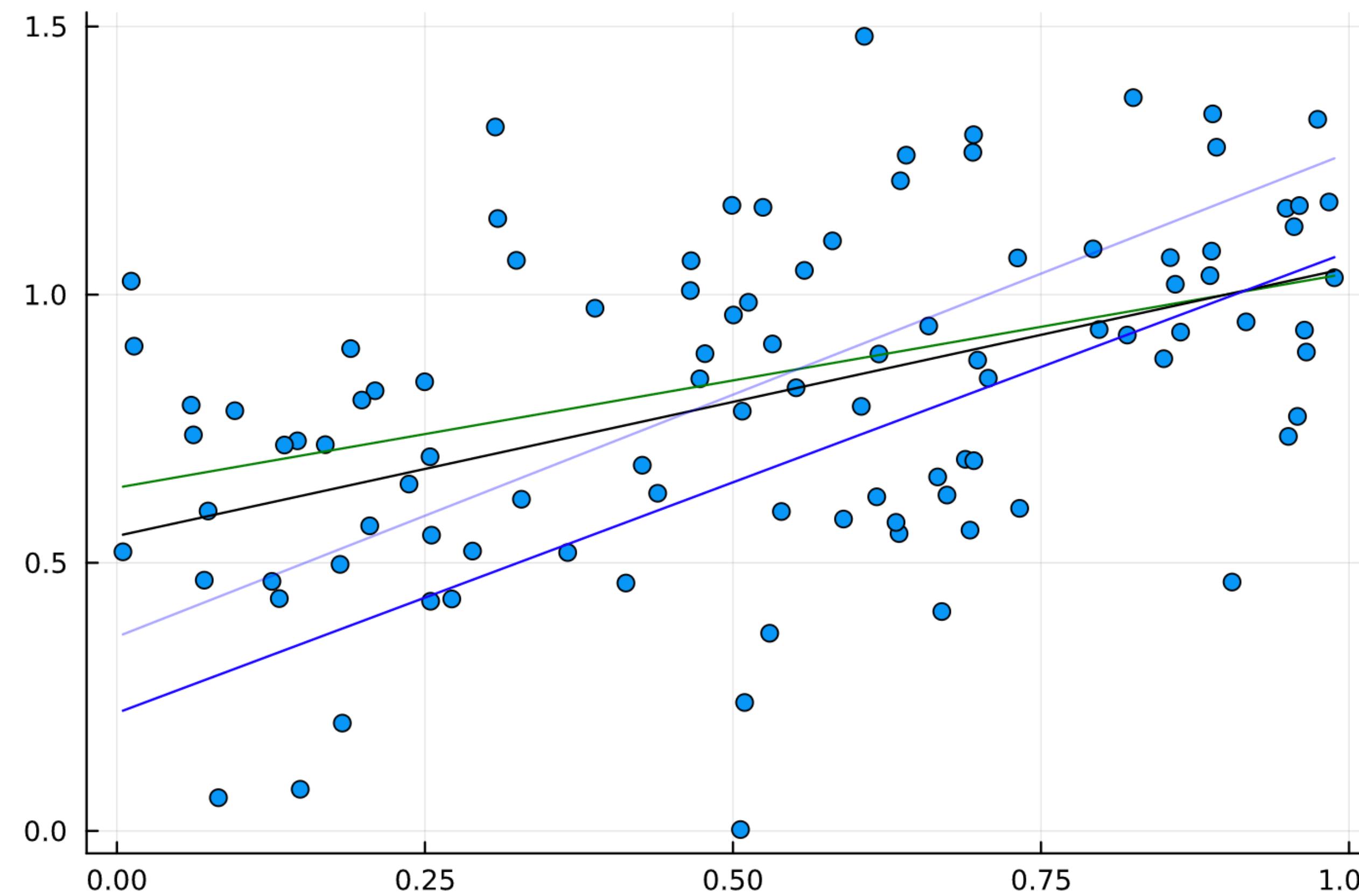
Linear Regression



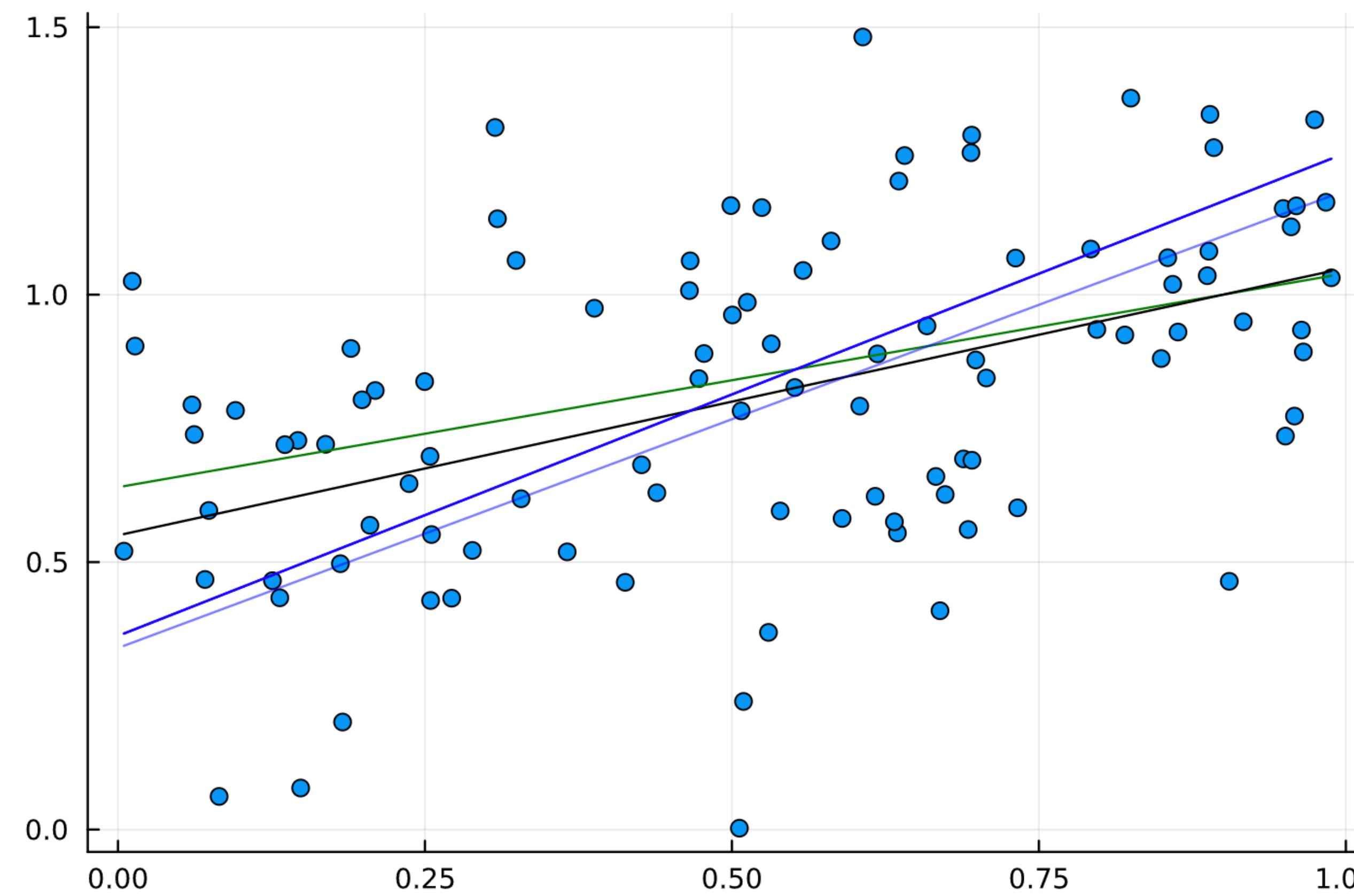
Linear Regression



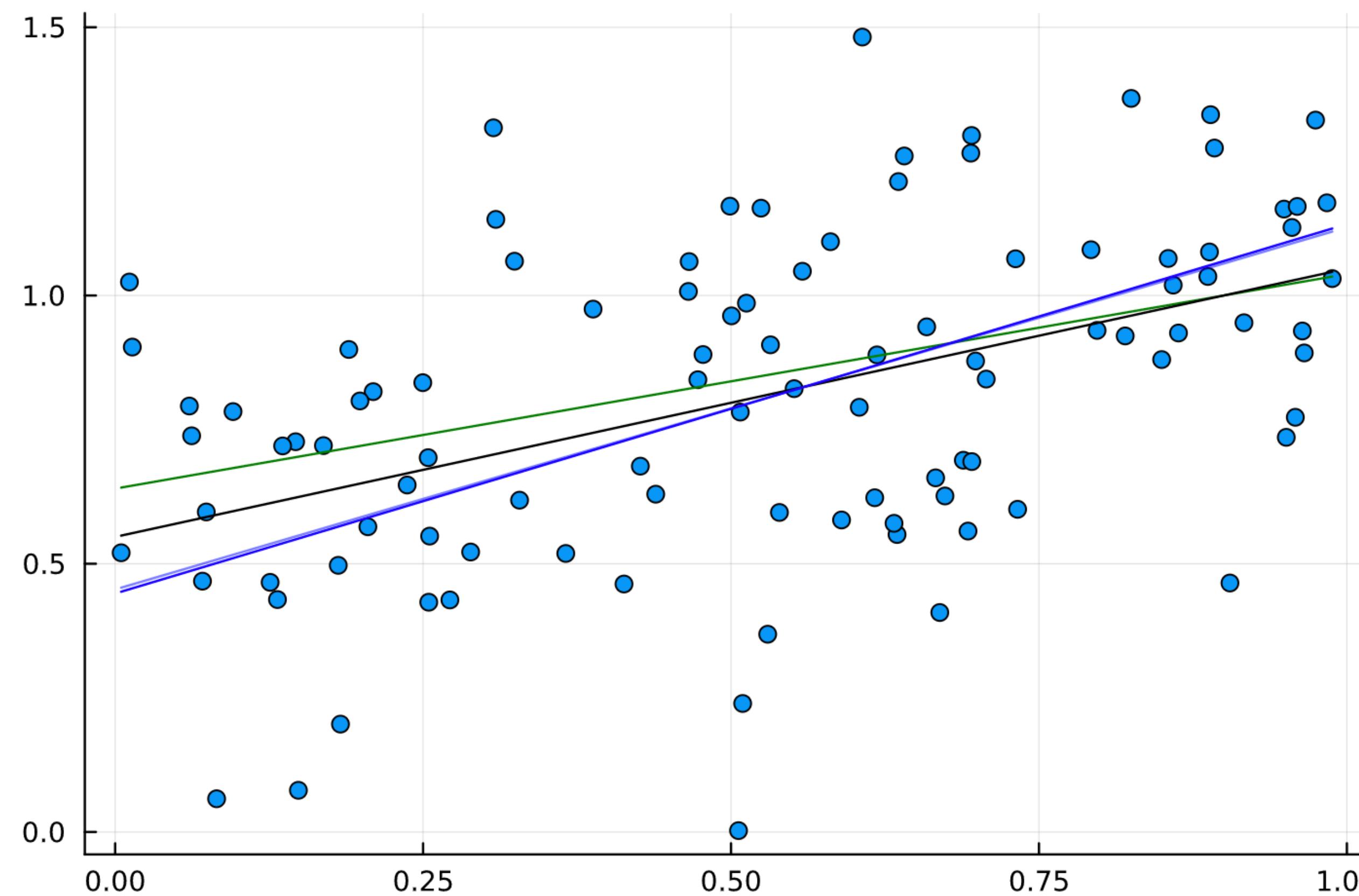
Linear Regression



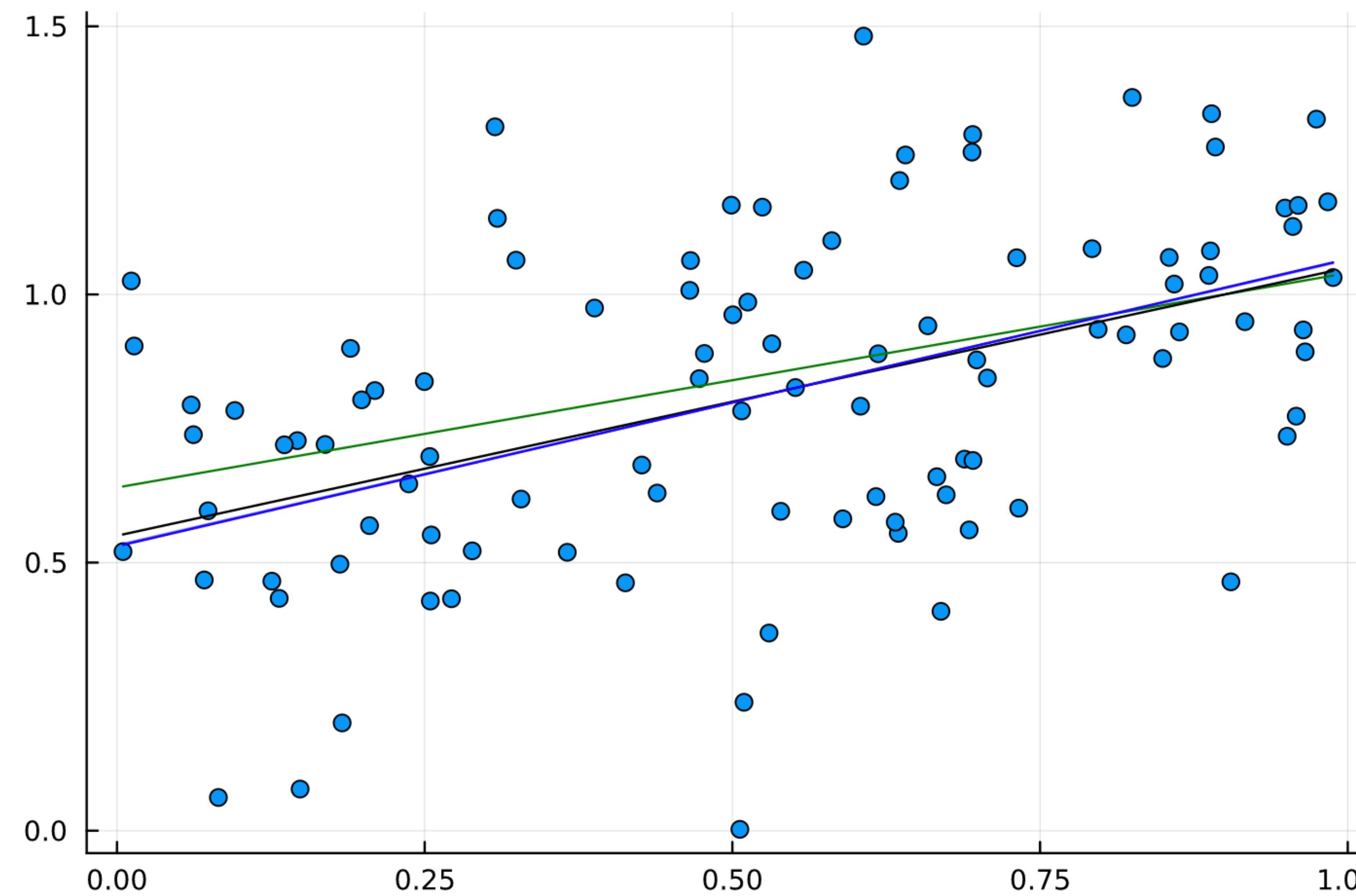
Linear Regression



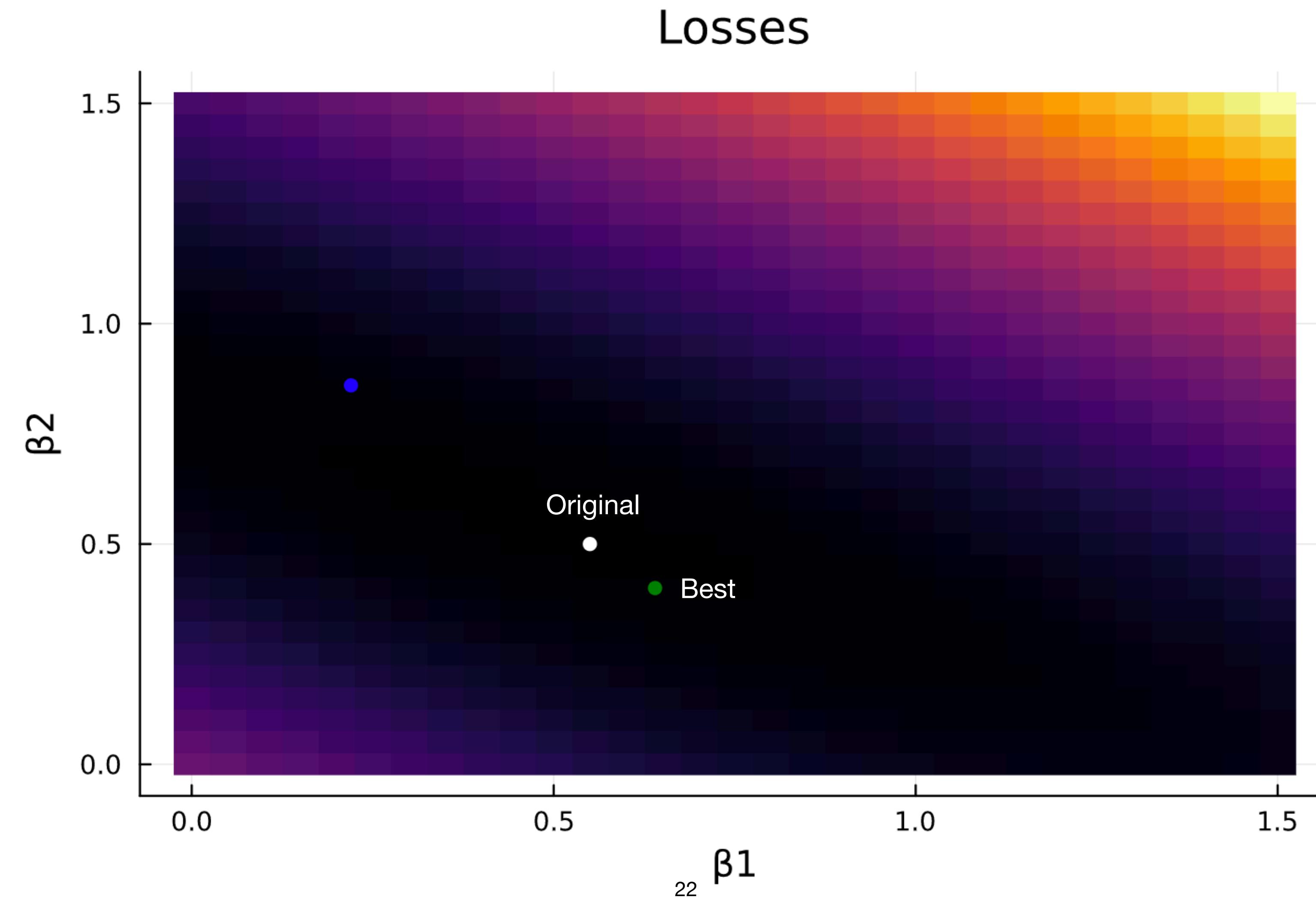
Linear Regression



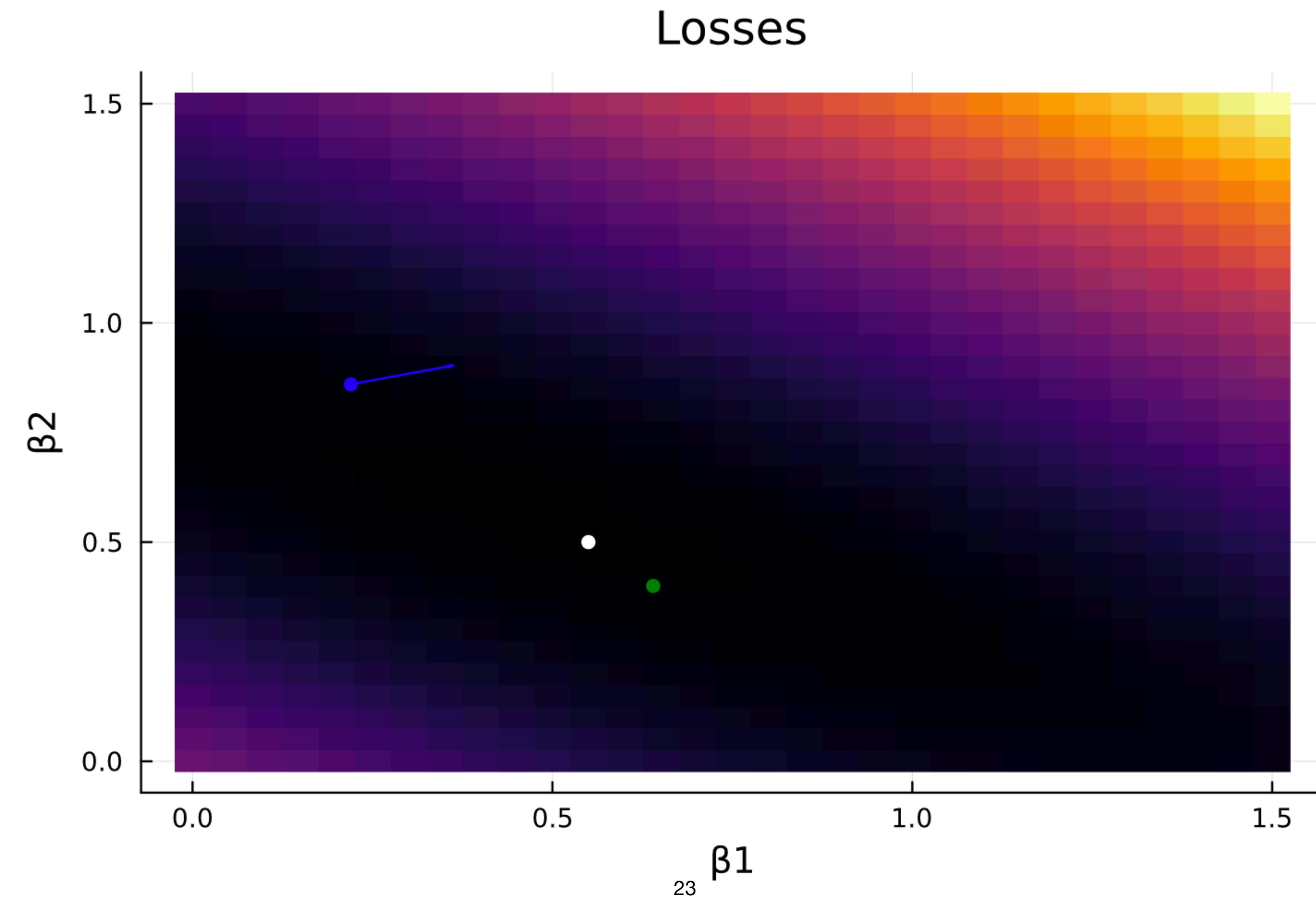
Linear Regression



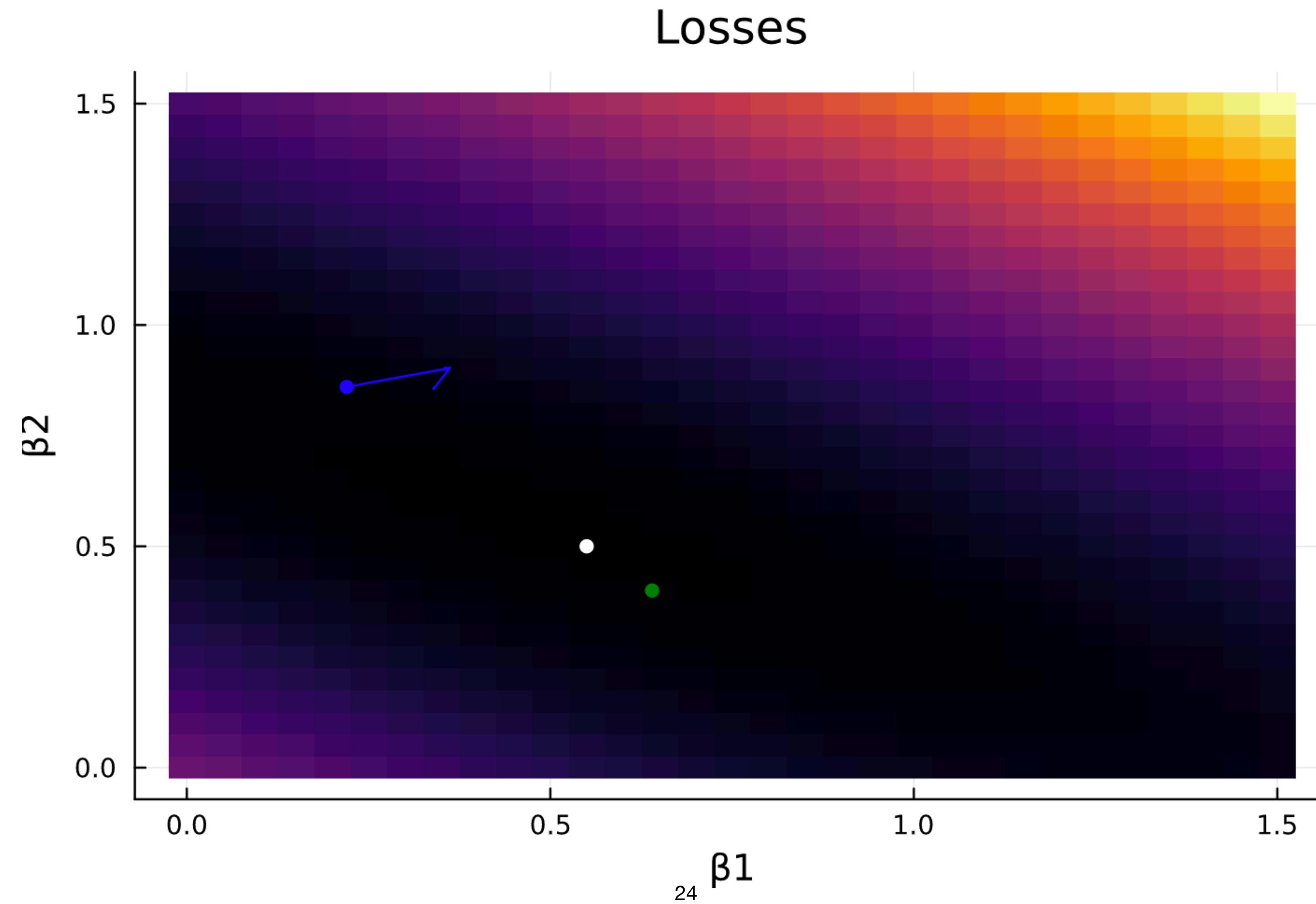
Linear Regression Loss



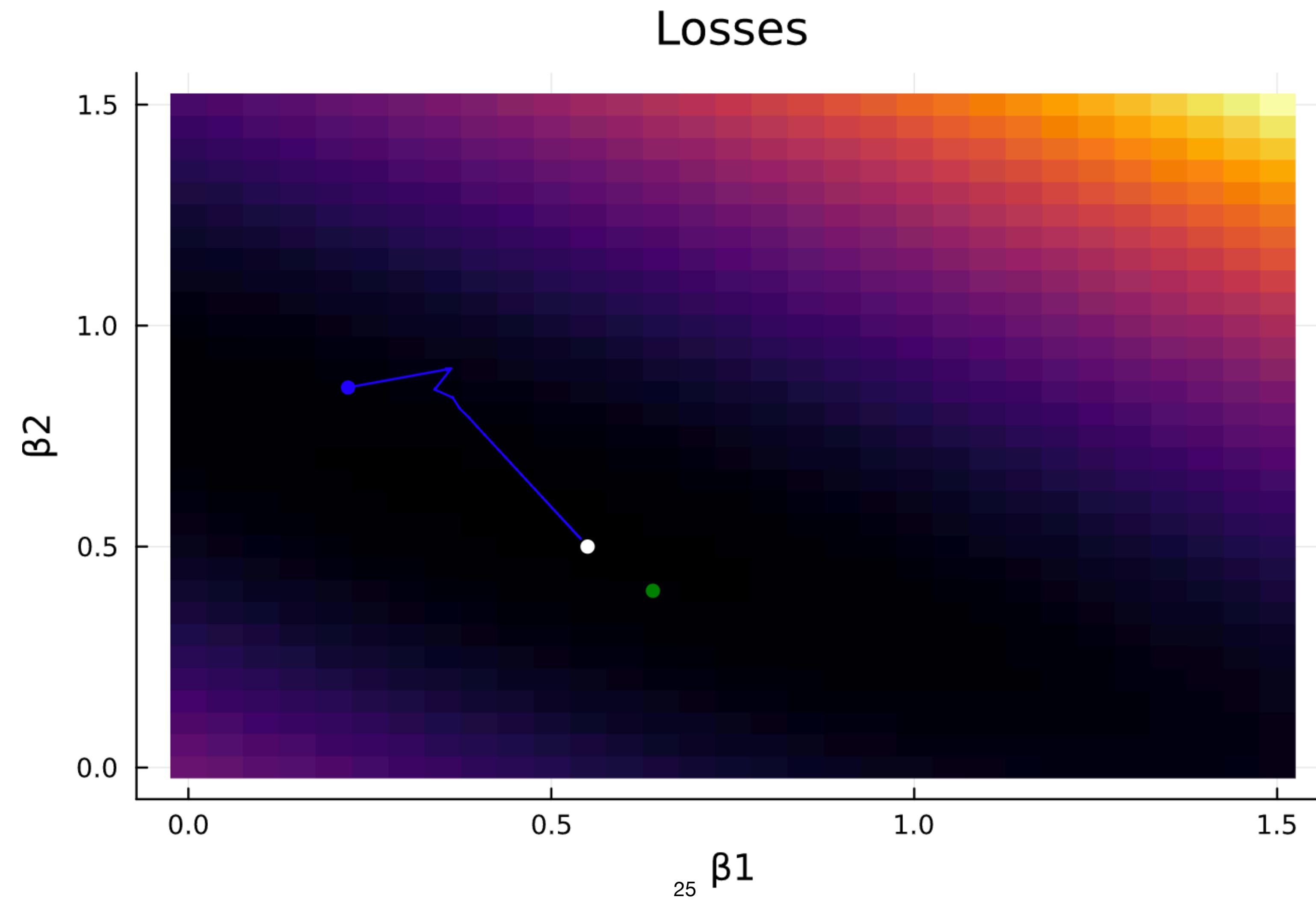
Linear Regression Loss



Linear Regression Loss



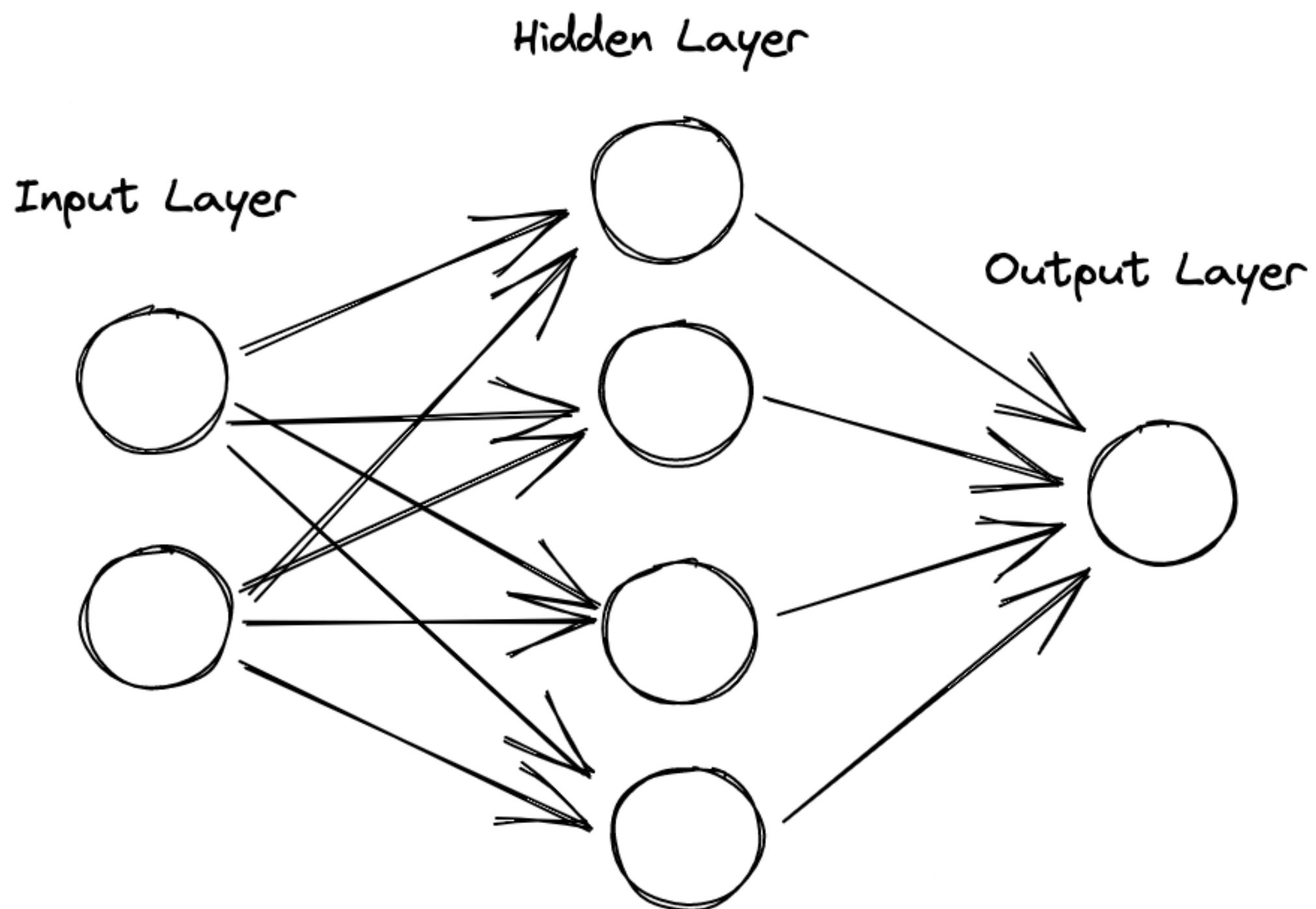
Linear Regression Loss



Linear Regression

```
# Initialize the coefficients ( $\beta$ ) with random values  
 $\beta = \text{rand}(\text{size}(X, 2))$   
  
for i in 1:N  
    # Calculate the gradient of the cost function with respect to  $\beta$   
     $\nabla J = X^T * (X * \beta - y)$   
  
    # Update  $\beta$  using gradient descent  
     $\beta -= \alpha * \nabla J$   
end
```

Feed Forward Model



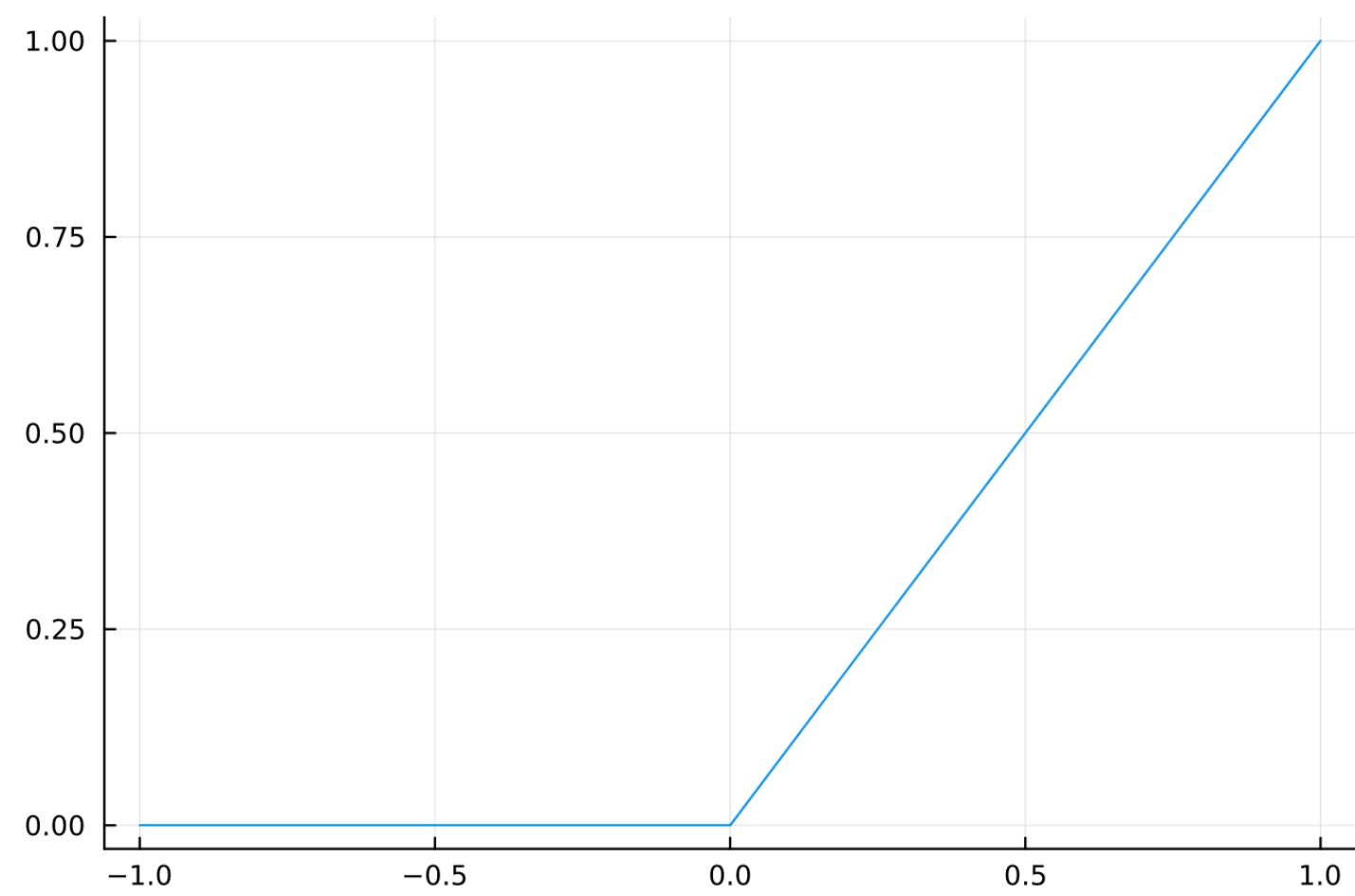
$$\begin{aligned} h_0 &= f_0(X \cdot W_0 + b_0) \\ h_i &= f_i(h_{i-1} \cdot W_i + b_i), \forall i \in \{1, \dots, H\} \\ h_f &= f_{out}(h_H \cdot W_{out} + b_{out}) = y \end{aligned}$$

Vocabulary

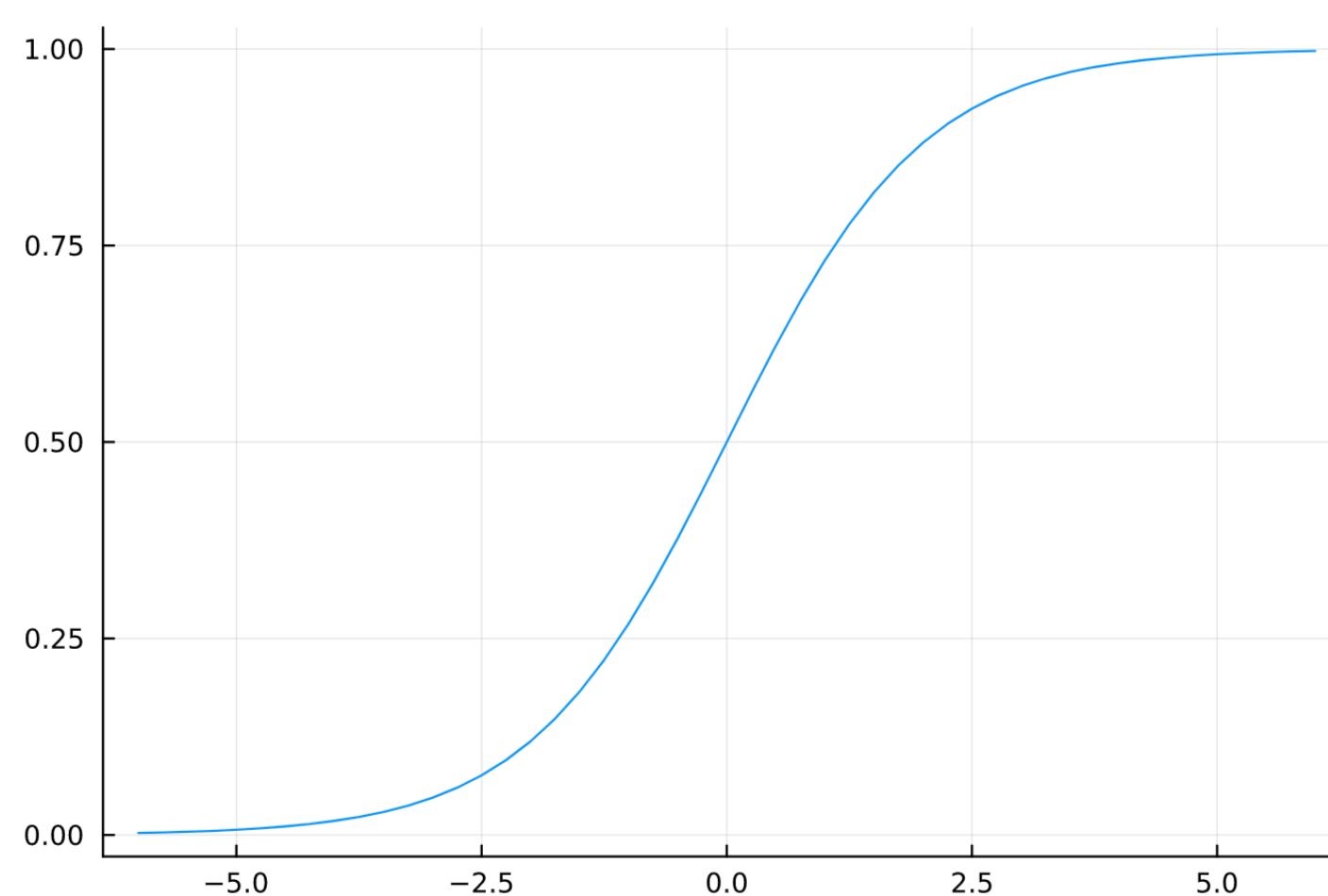
- **Epoch:** one entire pass over training data
- **Batch:** a small set of training data
- **Activation Function:** non-linear function applied over each element
- **Neurons:** where operations and activation functions happen

Activation Functions

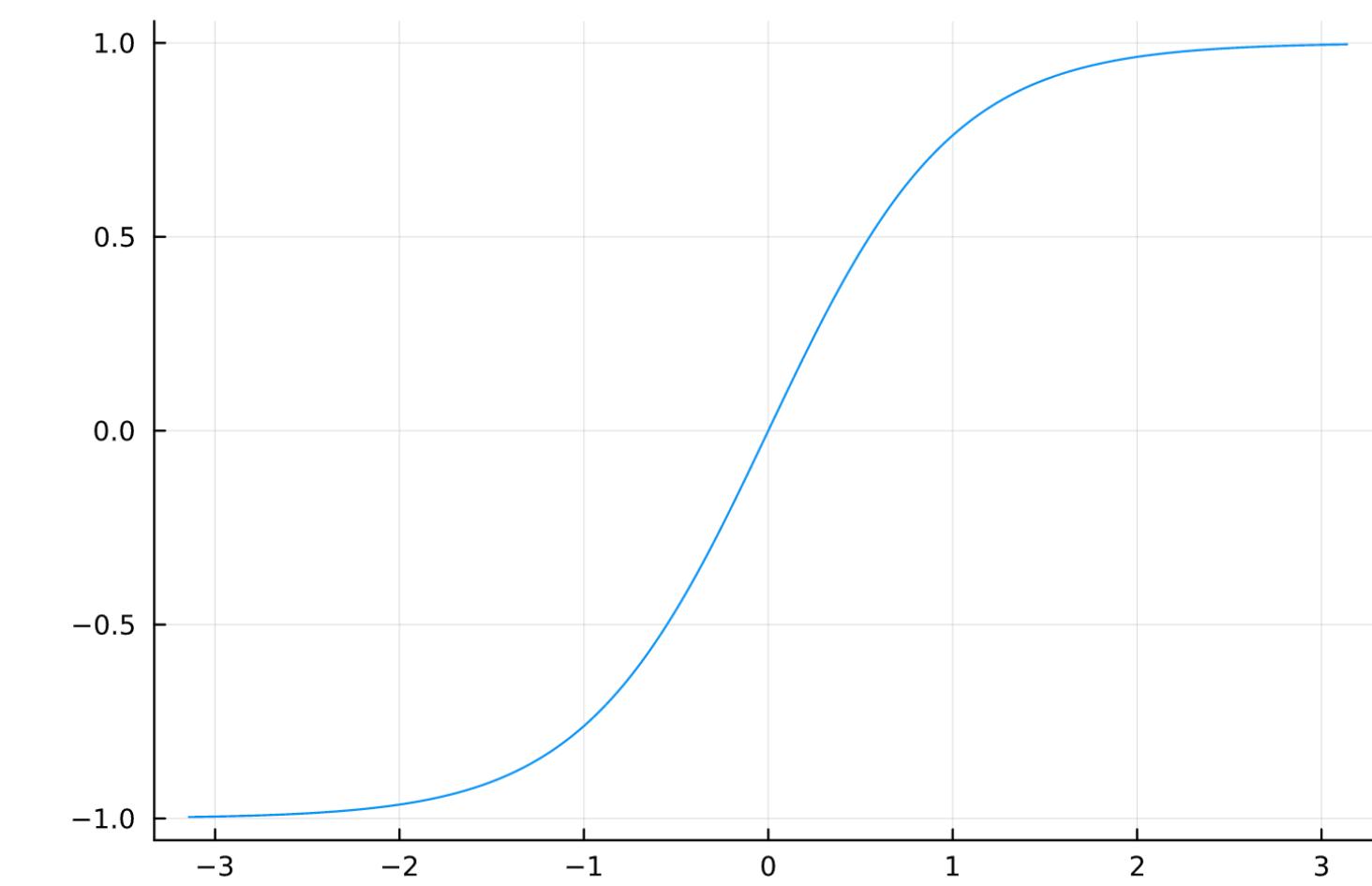
$$\text{ReLU} = \max(0, x)$$



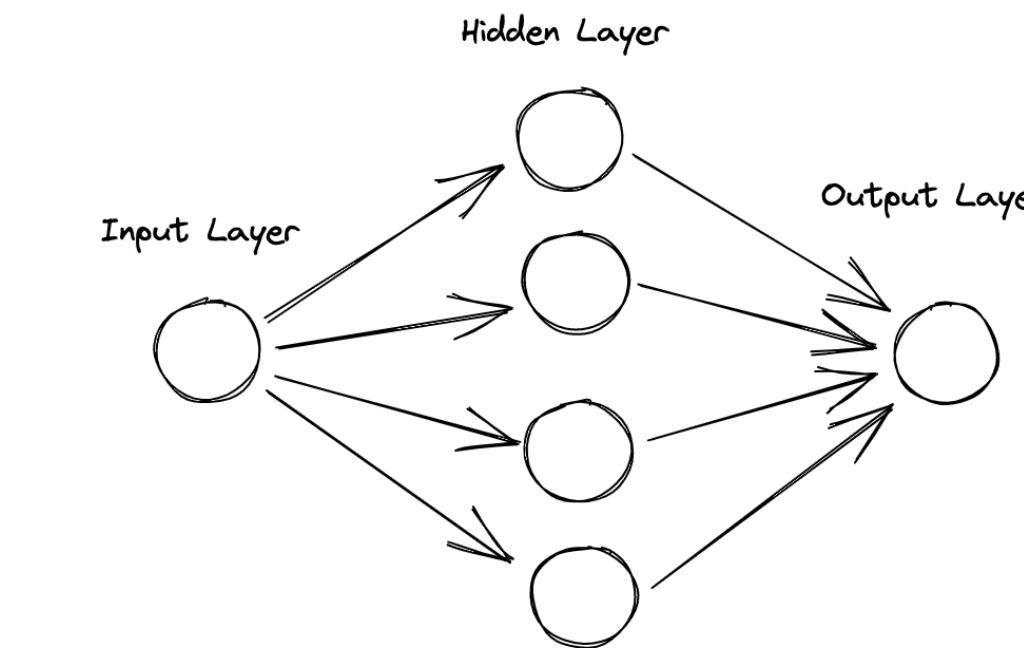
$$\sigma = \frac{1}{1 + e^{-x}}$$



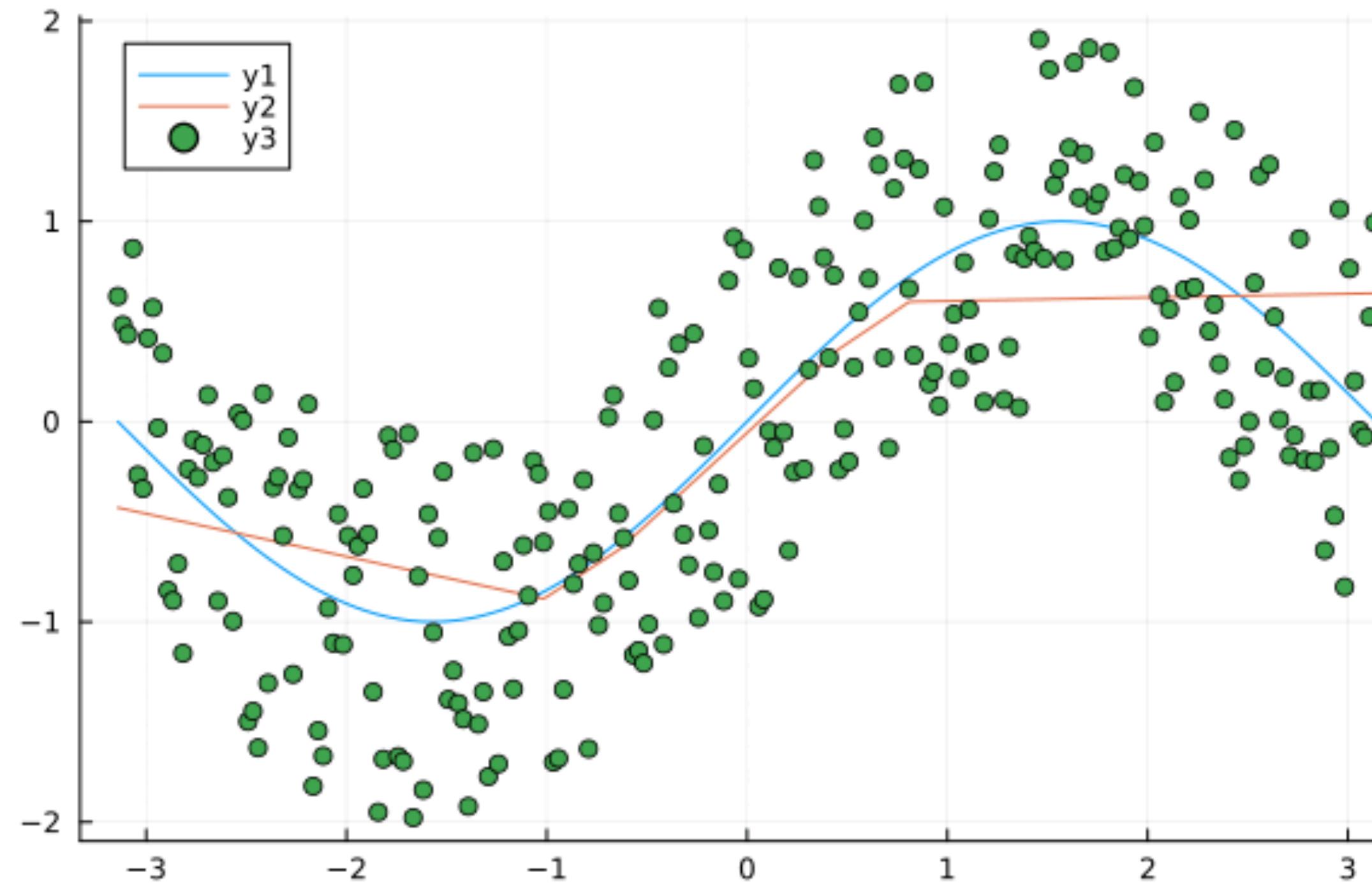
$$\tanh = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



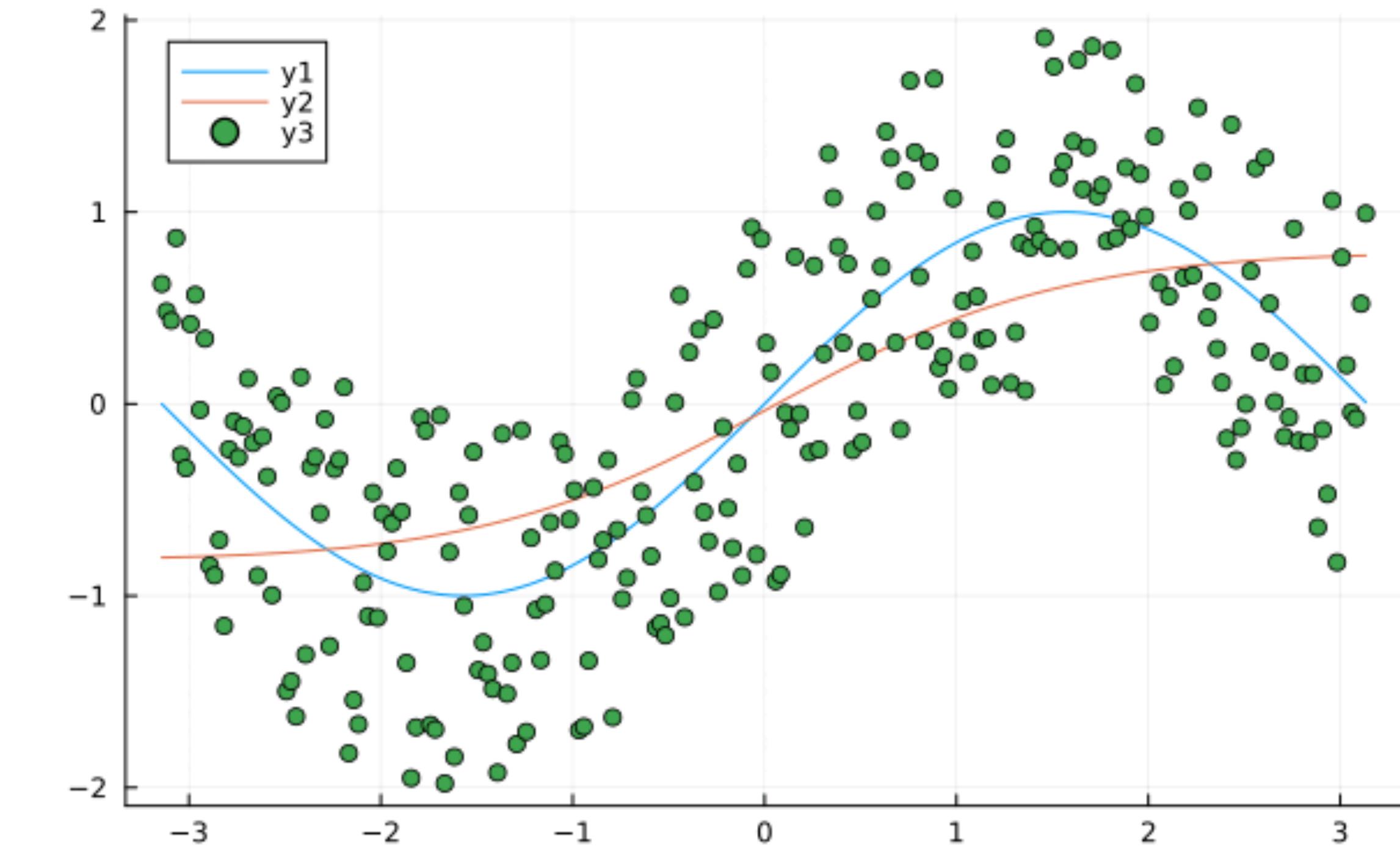
Breadth and Depth



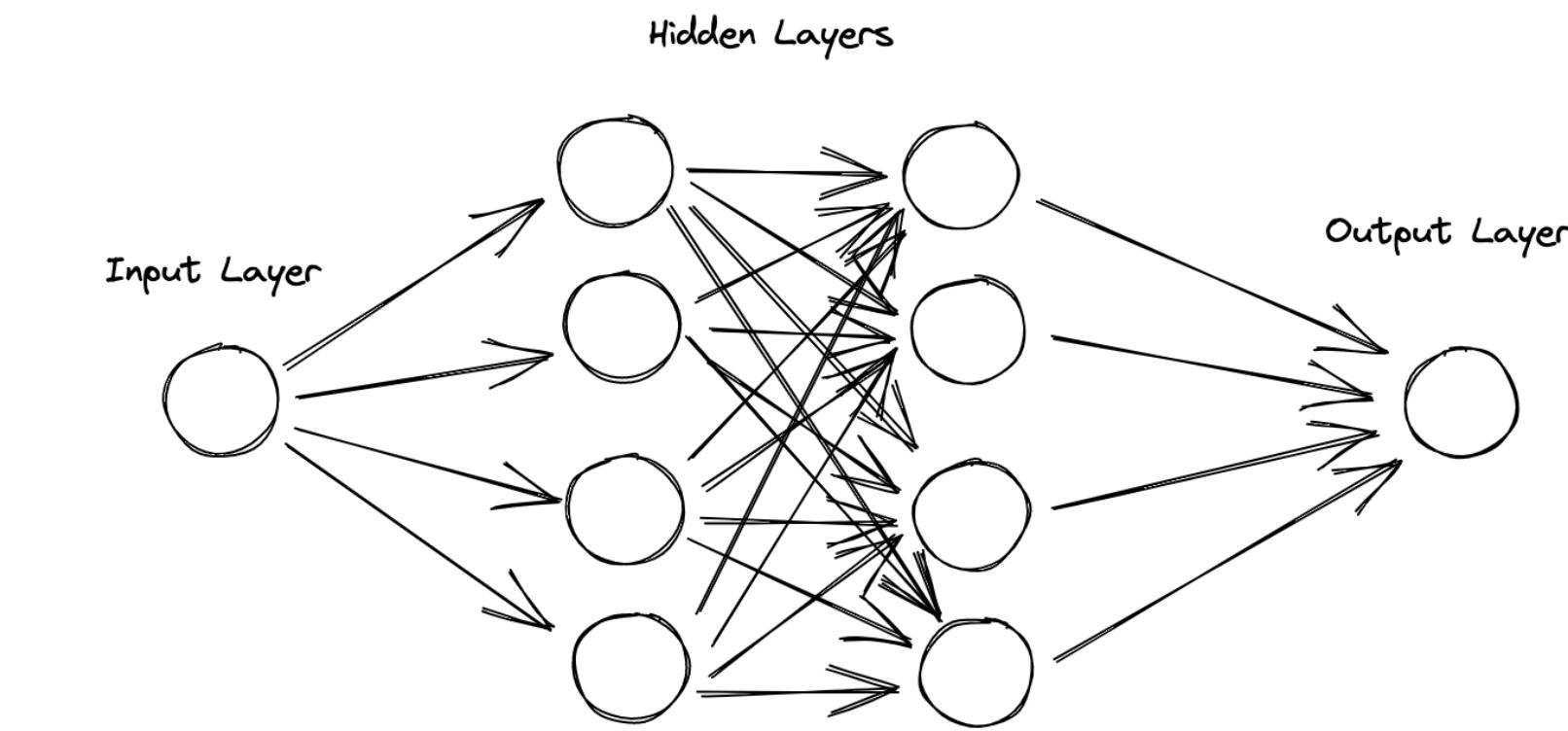
size=13 it=200 arch=((1, 4), (4, 1))



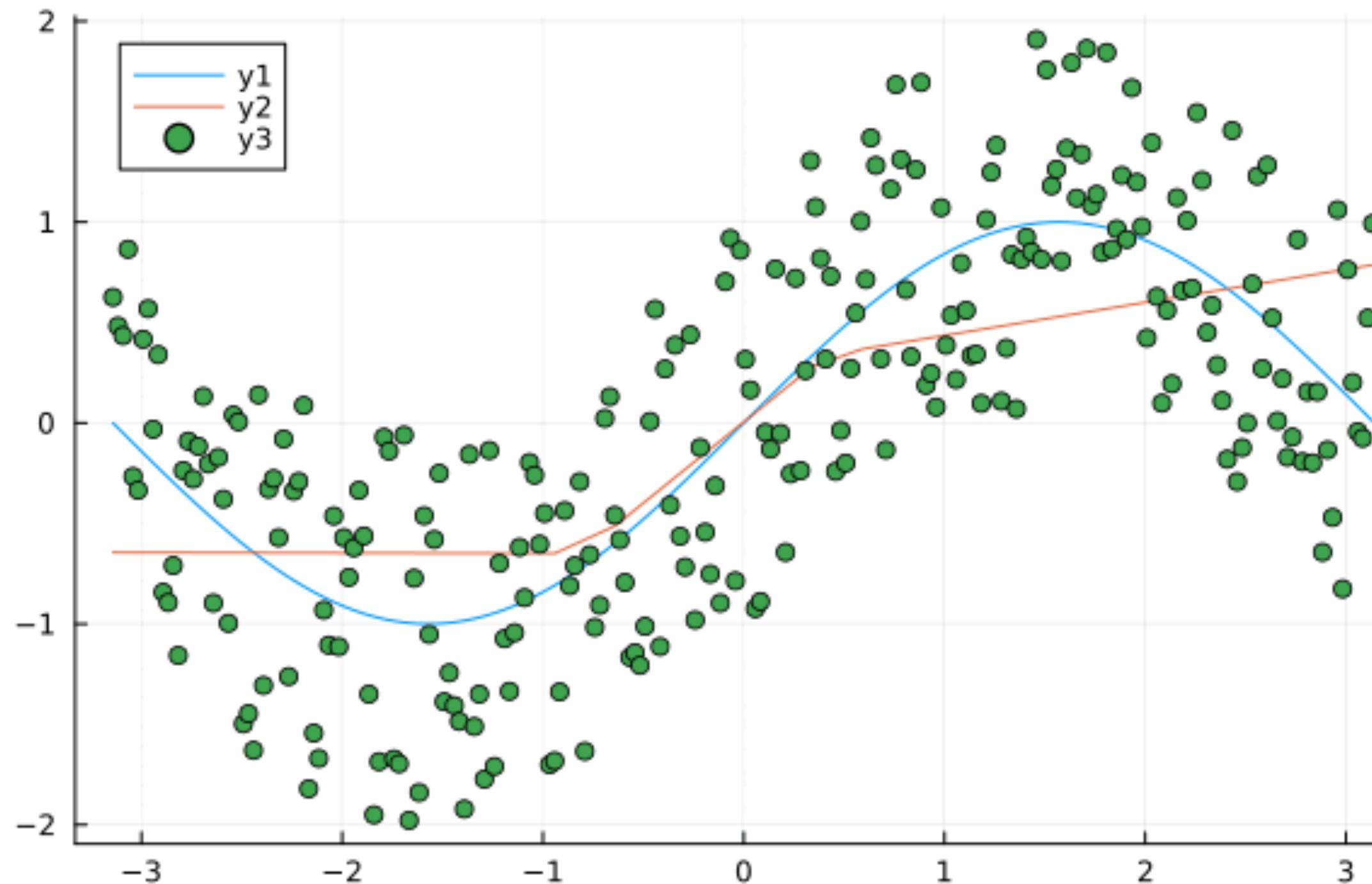
size=13 it=200 arch=((1, 4), (4, 1))



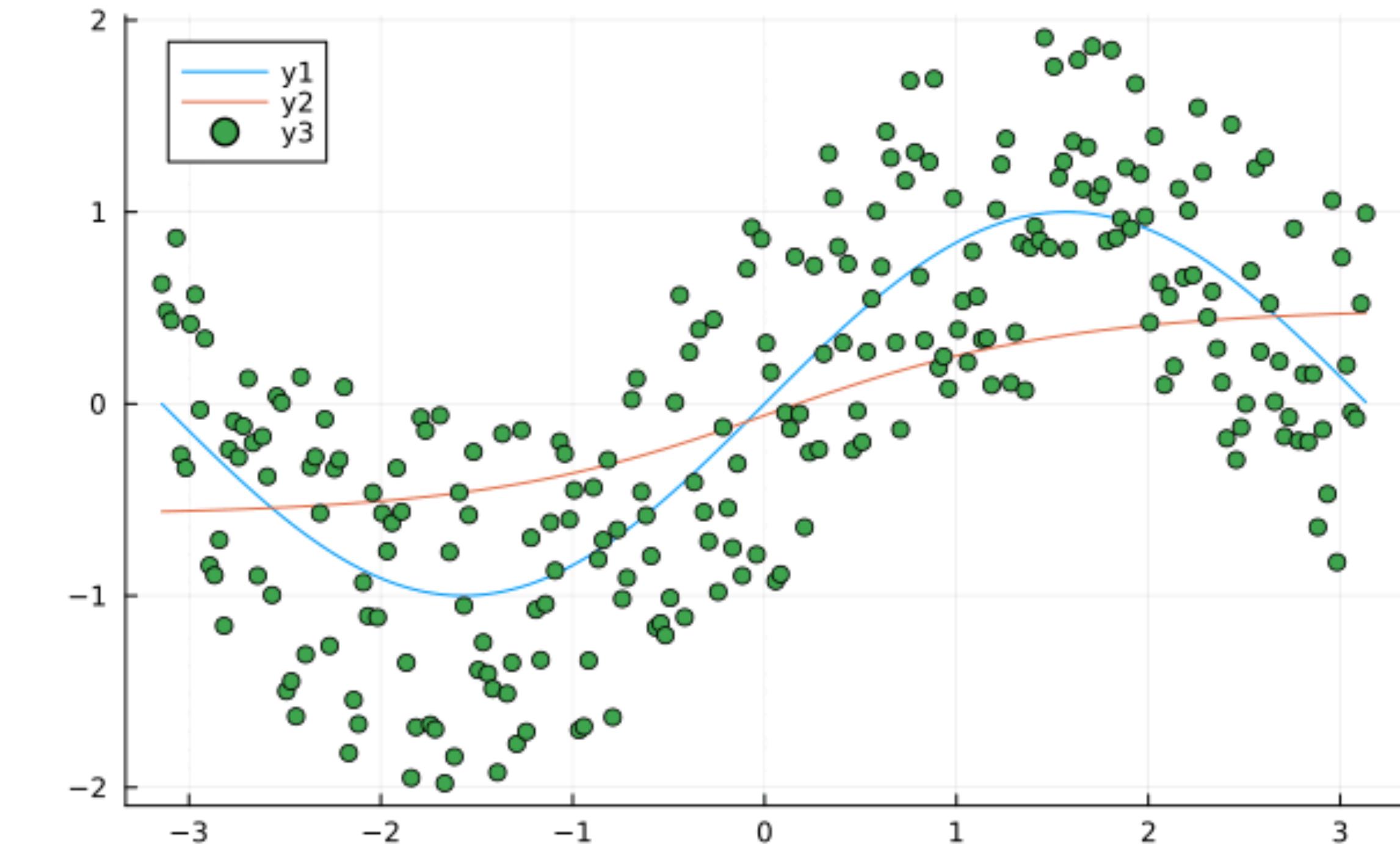
Breadth and Depth



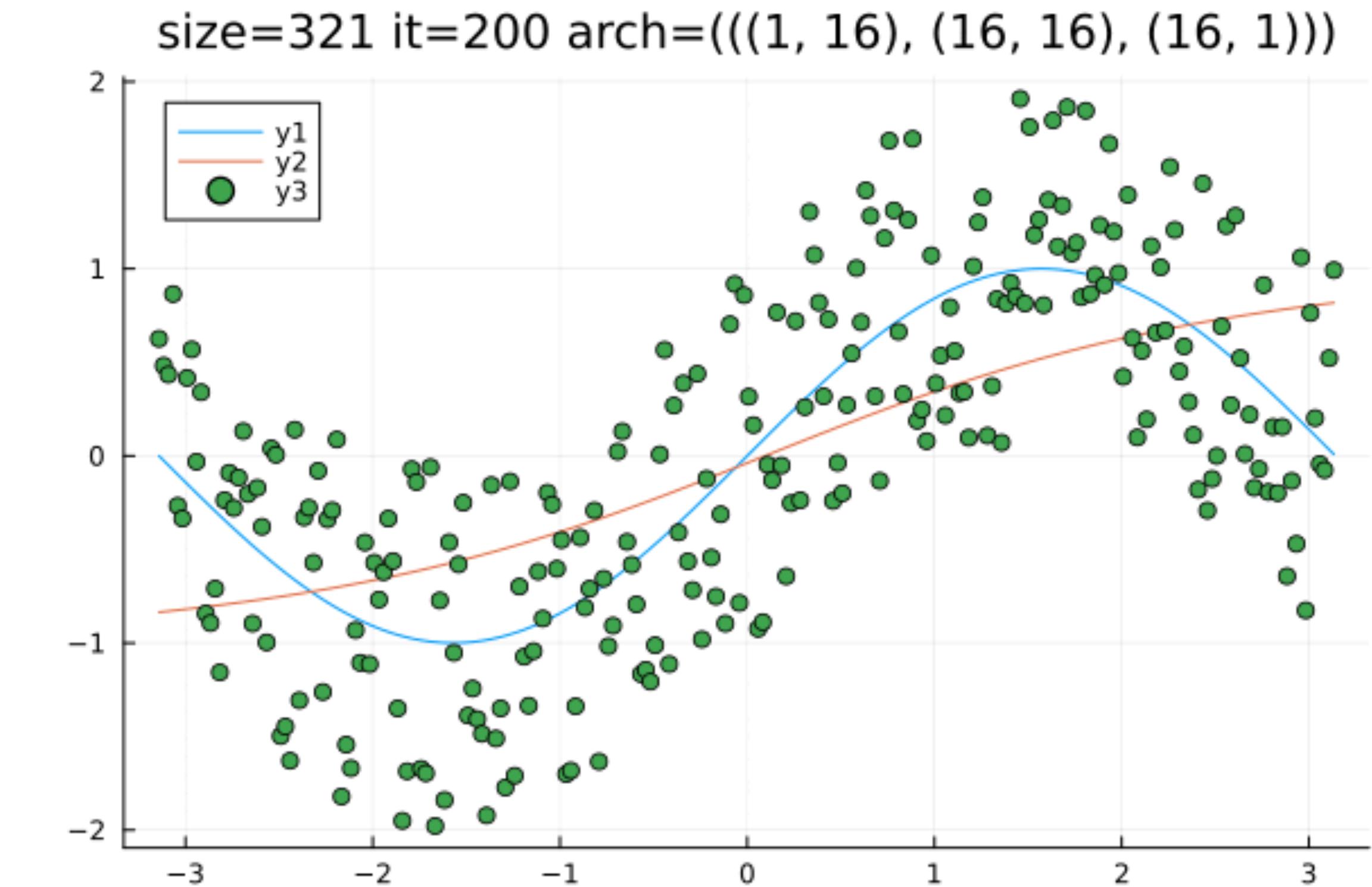
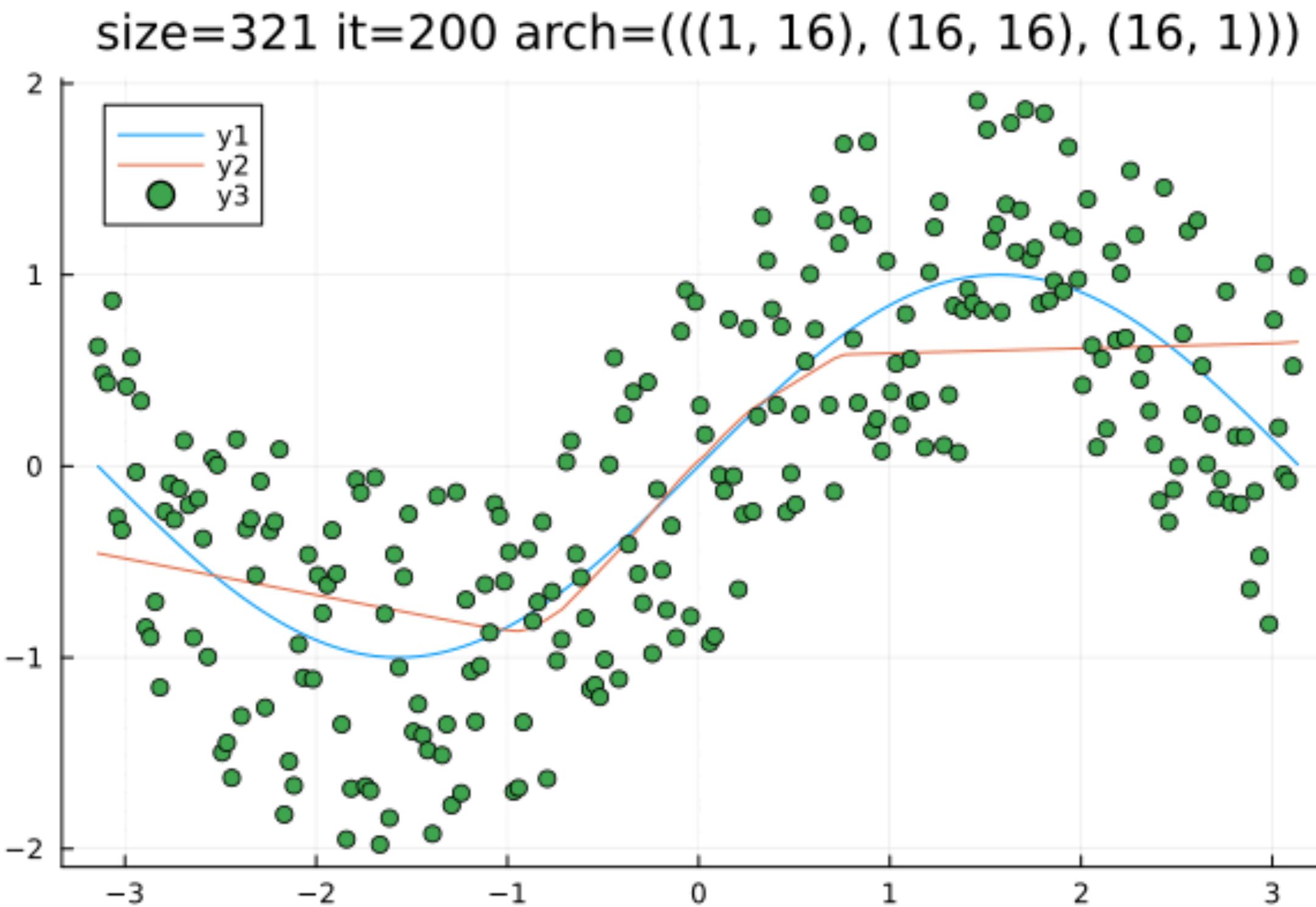
size=33 it=200 arch=((1, 4), (4, 4), (4, 1)))



size=33 it=200 arch=((1, 4), (4, 4), (4, 1)))

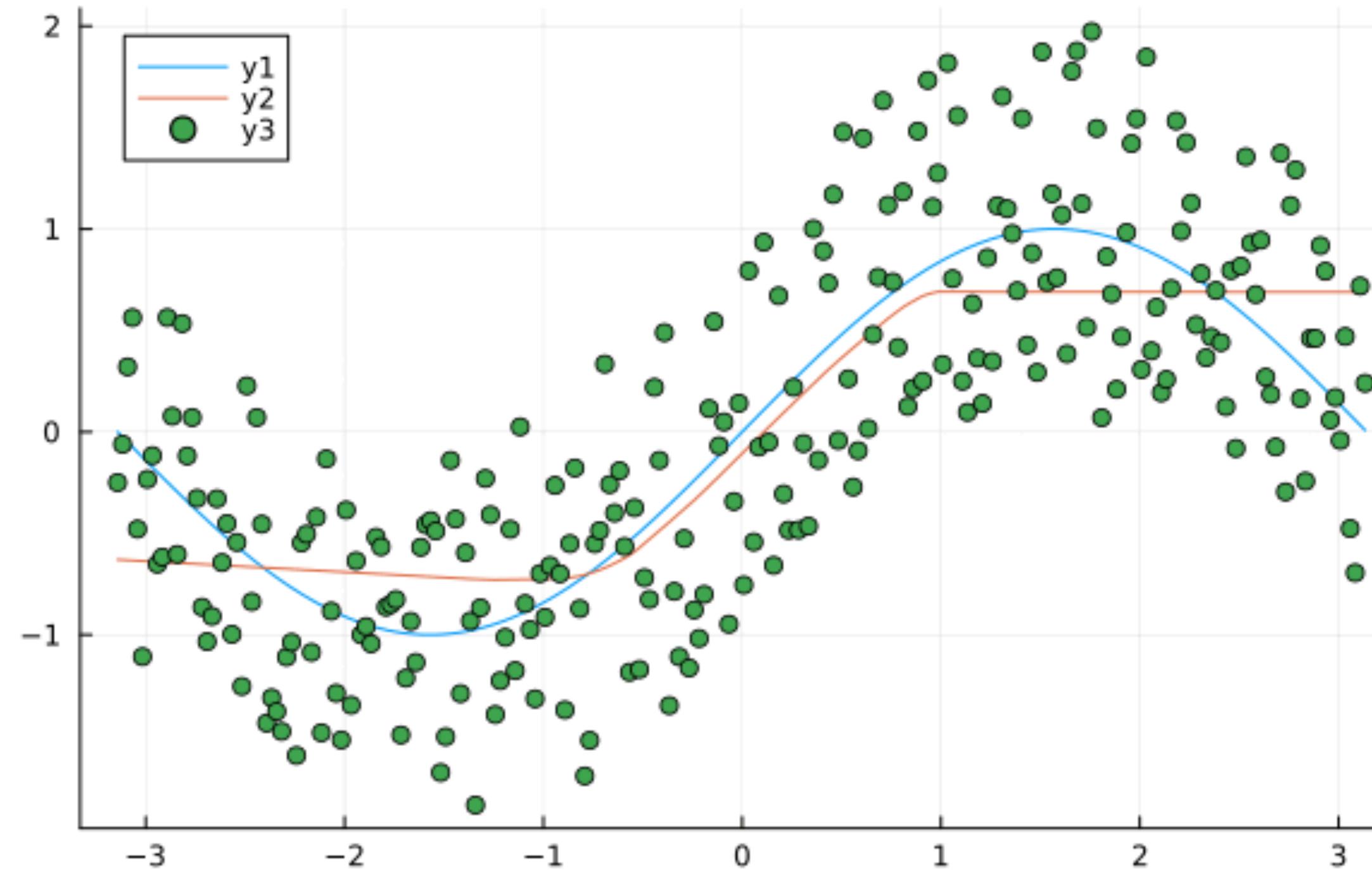


Breadth and Depth

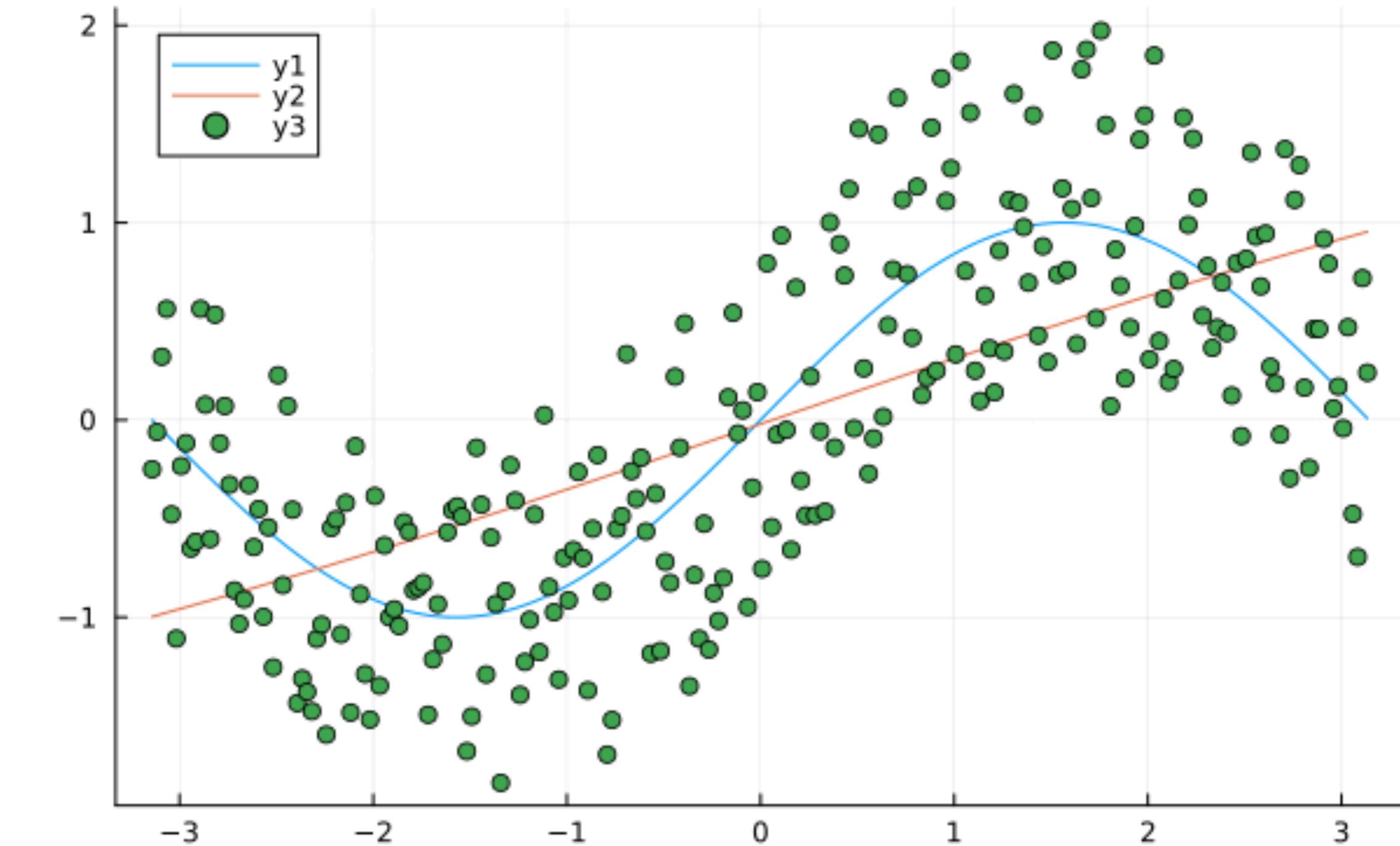


Breadth and Depth

size=193 it=200 arch=((1, 64), (64, 1)))



size=193 it=200 arch=((1, 64), (64, 1)))



Implementation

using Flux # Flux is the Neural Network Package for Julia (equivalent to tf.Keras and PyTorch)

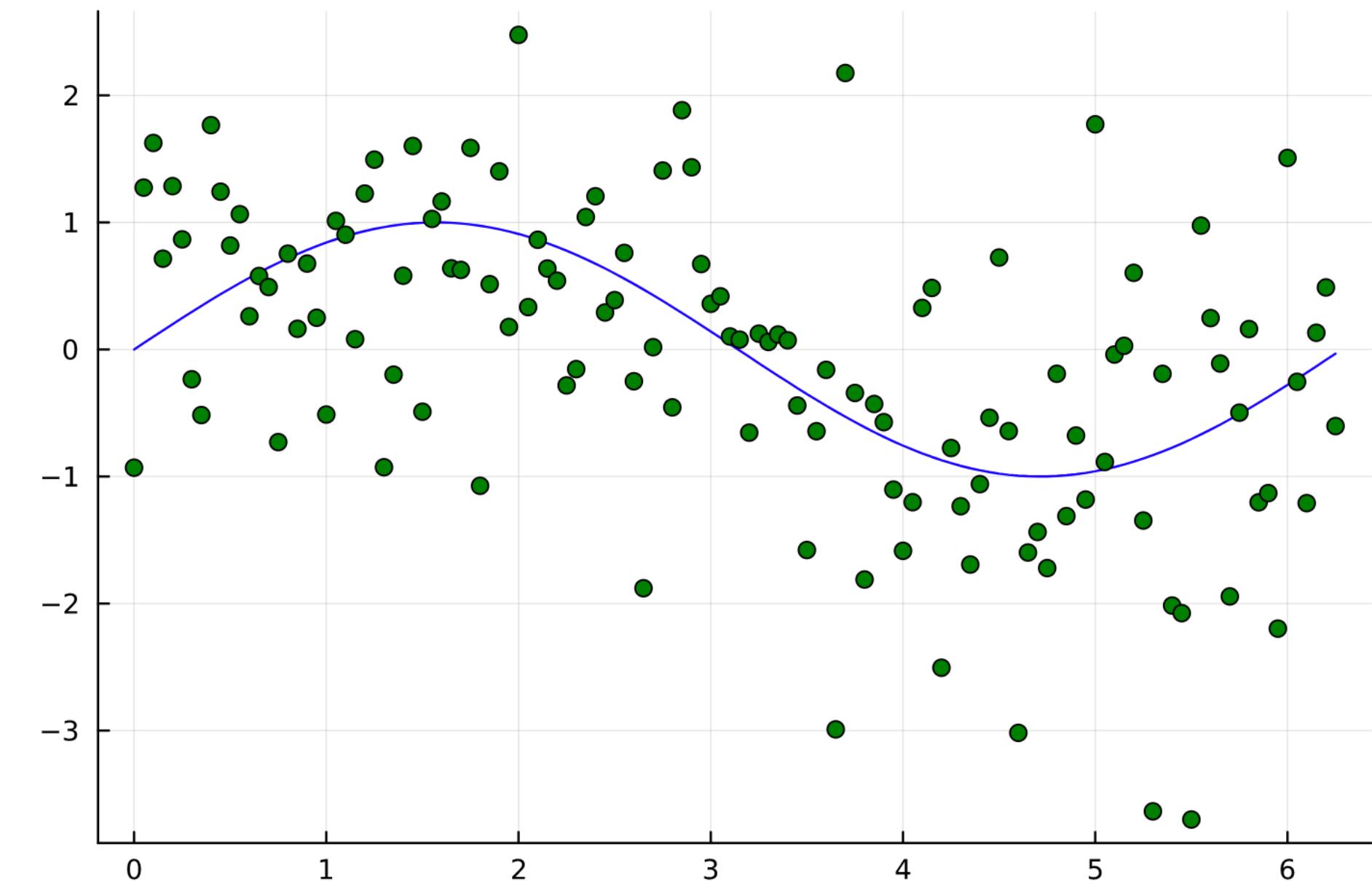
```
# Define the independent variable x and dependent variable y
```

```
x = collect(0:0.05:2π)
```

```
y = sin.(x) + ε
```

```
# Reshape x into a matrix with one row
```

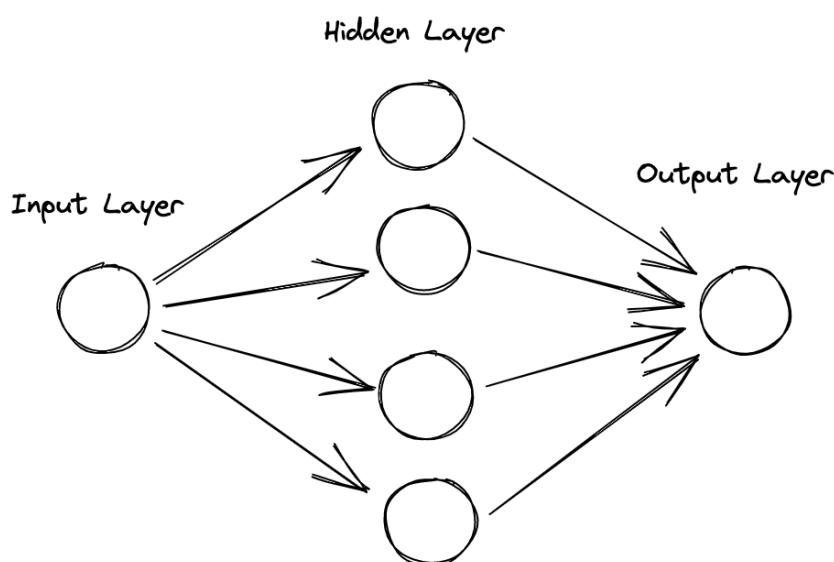
```
X = reshape(x, 1, :) # Flux expects size(input, 2) = batch size
```



Implementation

```
# Define the model as a chain of layers with a ReLU  
# activation function in the hidden layer
```

```
hidden_size = 4  
model = Chain(  
    Dense(1, hidden_size, relu),  
    Dense(hidden_size, 1)  
)
```



```
# Define the loss function as the mean squared error  
loss(X, y) = Flux.mse(model(X), y)
```

$$\hat{y} = \max(X \cdot W_{input} + b_{input}, 0) \cdot W_{out} + b_{out}$$

$\begin{matrix} N \times 1 \\ N \times 2 & 2 \times 4 & 1 \times 4 \\ & N \times 4 & \\ & & 4 \times 1 & 1 \times 1 \end{matrix}$

$N \times 1$

$$J(W_i) = \frac{1}{2} \sum_{i=1}^N (y - \hat{y})^2 =$$
$$\therefore f_{input}(X) = \max(X \cdot W_{input} + b_{input}, 0)$$
$$\rightarrow \frac{\partial J}{\partial W_{out}} = f_{input}(X)^T \cdot (f_{input}(X) - y)$$

Implementation

```
# Define the optimizer as stochastic gradient
```

```
# descent with a learning rate of 0.01
```

```
opt = Descent(0.01)  $\eta = 0.01$ 
```

```
# Train the model for 100 epochs
```

```
for i in 1:1000
```

```
    Flux.train!(loss, model, [(X,y)], opt)
```

```
end
```

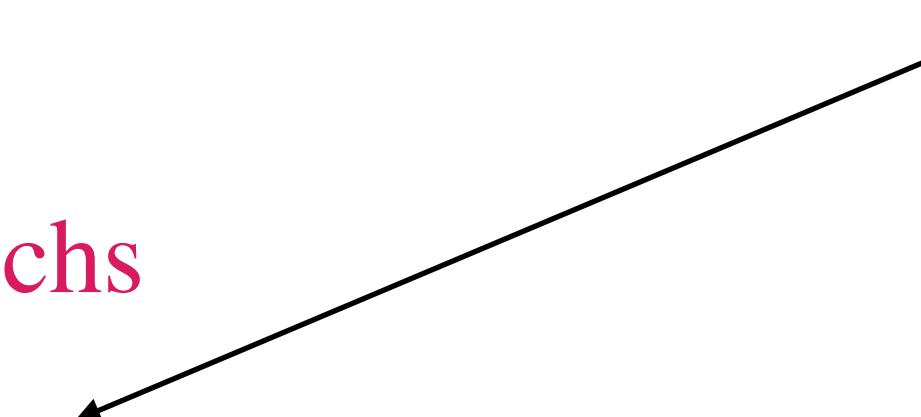
```
# Make predictions using the trained model
```

```
ŷ = model(X)
```

Implementation

```
# Train the model for 100 epochs
for i in 1:1000
    Flux.train!(loss, model, [(X,y)], opt)
end
```

What about technical
constraints, e.g. memory?



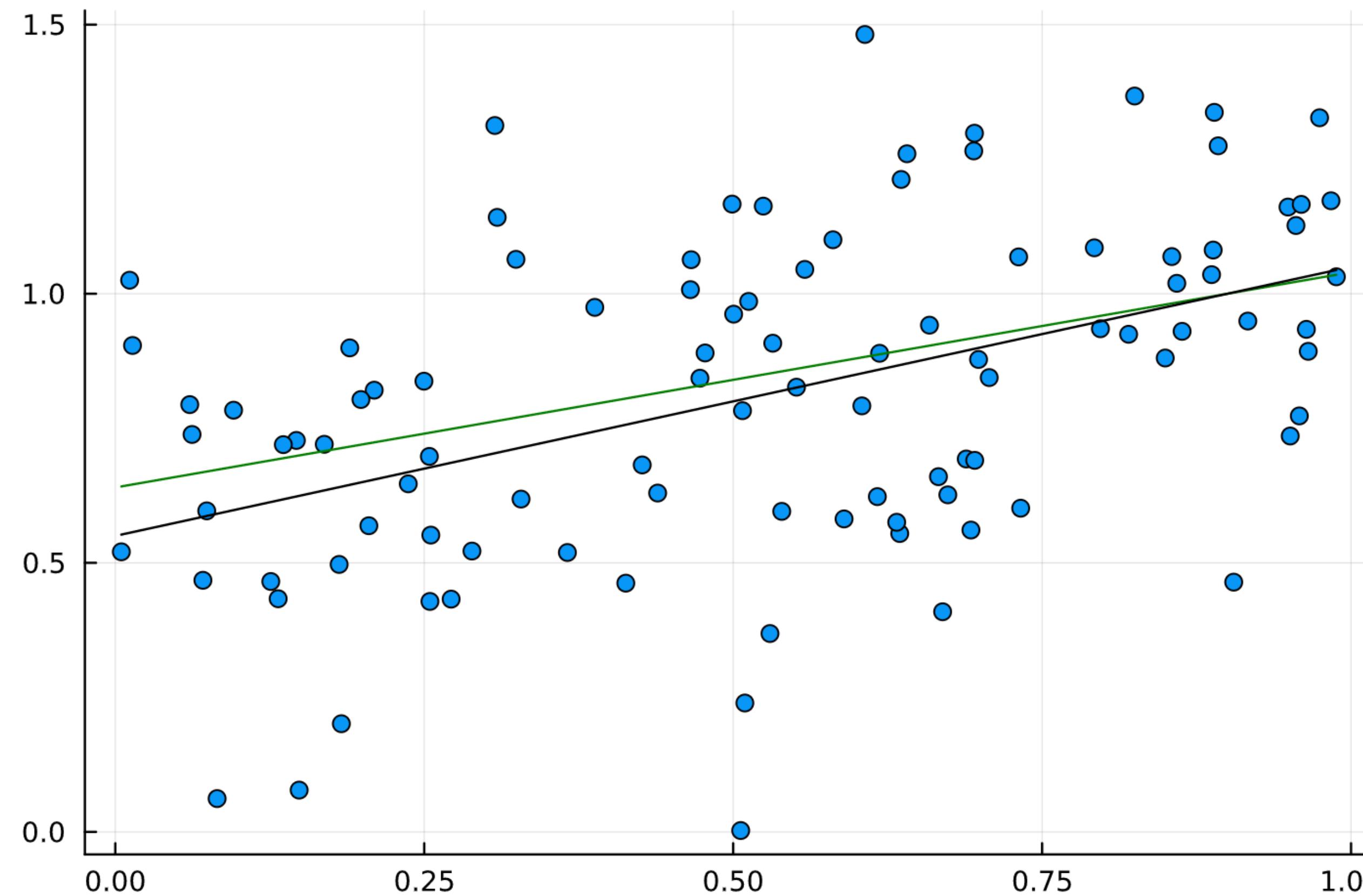
Implementation

- MNIST has 50.000 28x28 PB (float32) images: $14.954 \text{ MiB} \approx 15 \text{ GB}$
- The network used in the example has 16.938 parameters: $2.023 \text{ KiB} \approx 2 \text{ MB}$
- (consider that each parameter also has partial derivative)
- How can we train a model that we can not fit into memory?

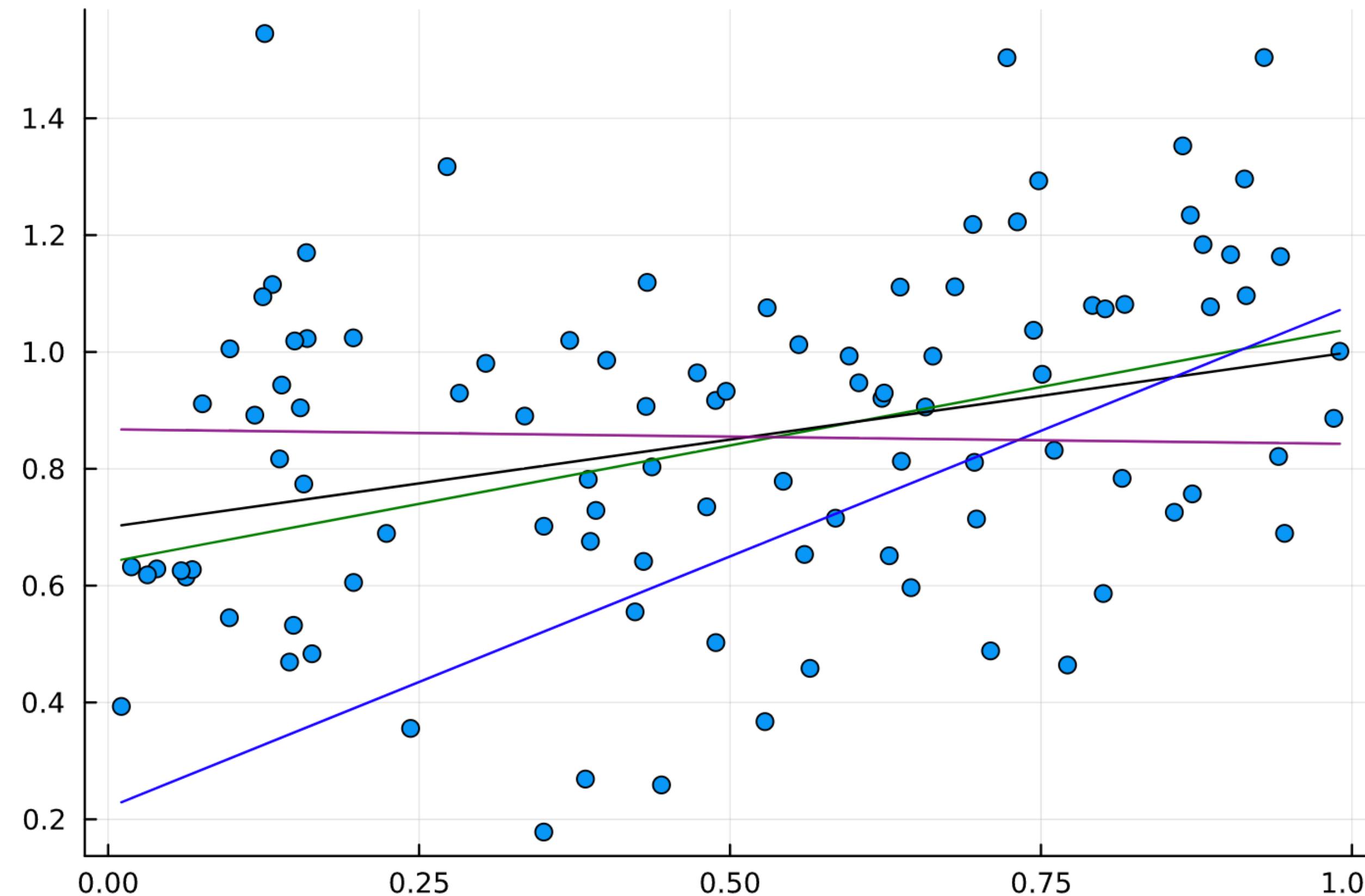
Stochastic Gradient Descent

$$\begin{aligned}\beta &= \beta - \eta \cdot \frac{\delta J}{\delta \beta} \\ &= \beta - \eta \cdot X^T \cdot (X \cdot \beta - y) \\ &= \beta - \frac{\eta}{n} \cdot \sum_{i \in \Omega} X_i^T \cdot (X_i \cdot \beta - y_i)\end{aligned}$$

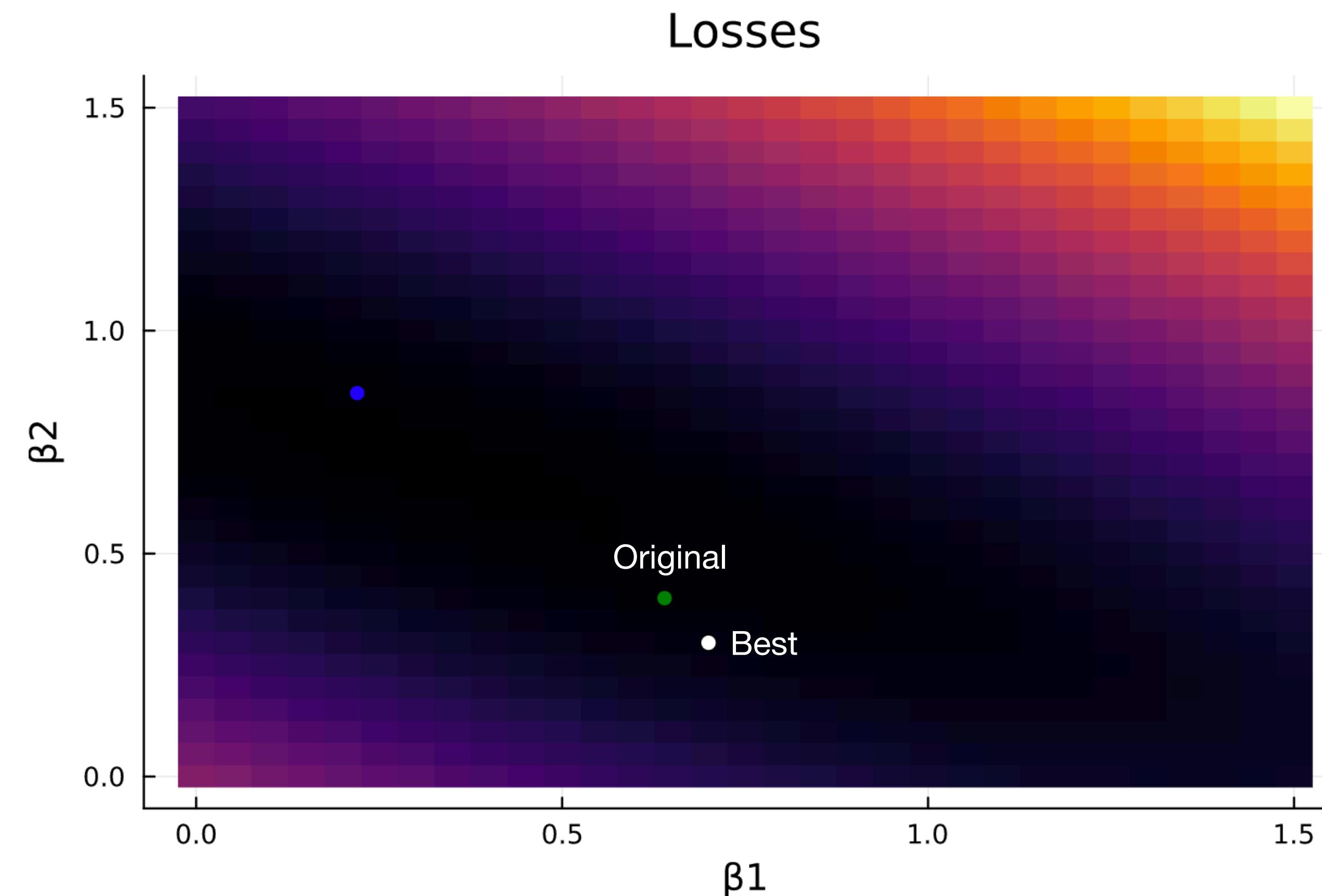
Stochastic Gradient Descent



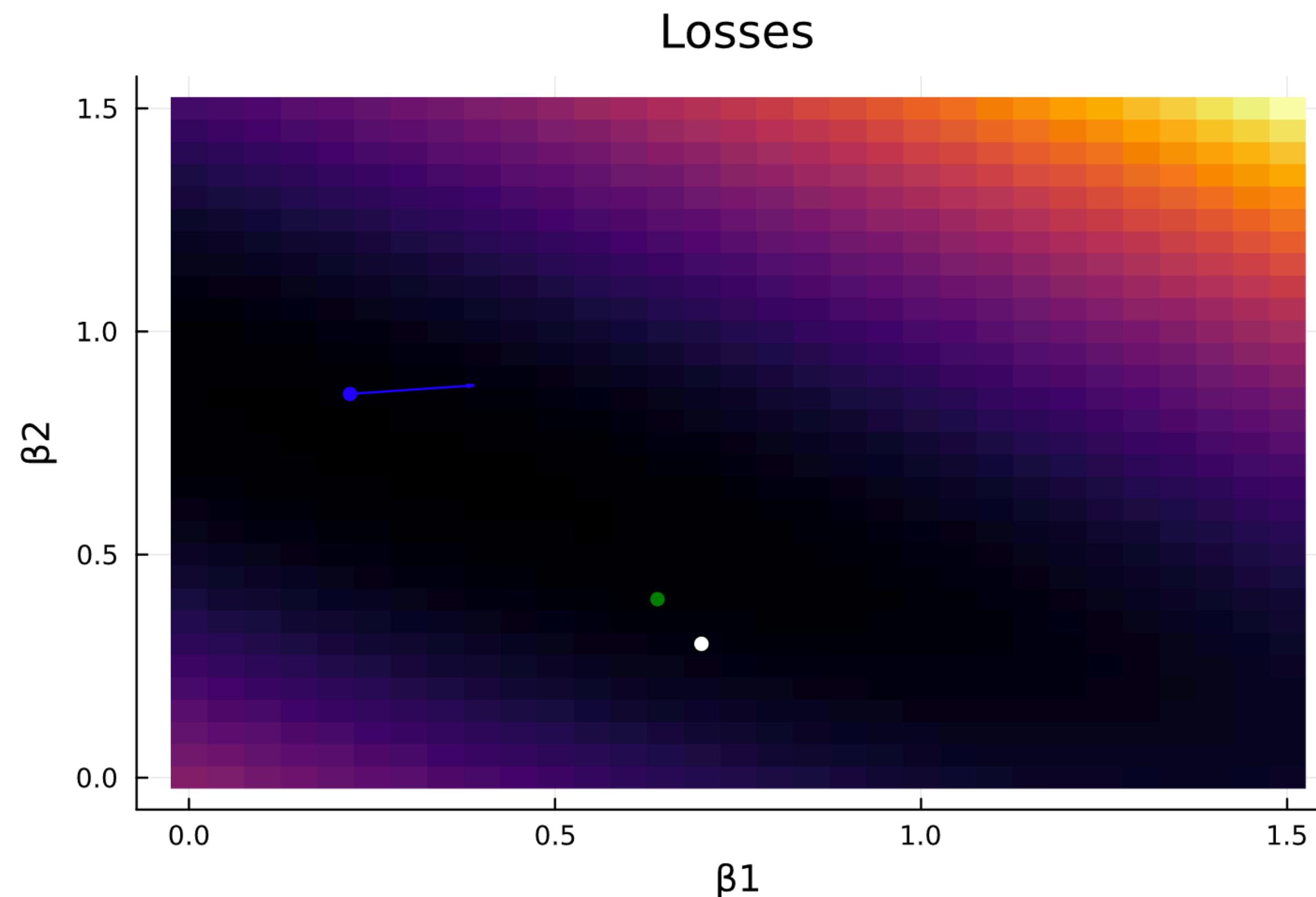
Stochastic Gradient Descent



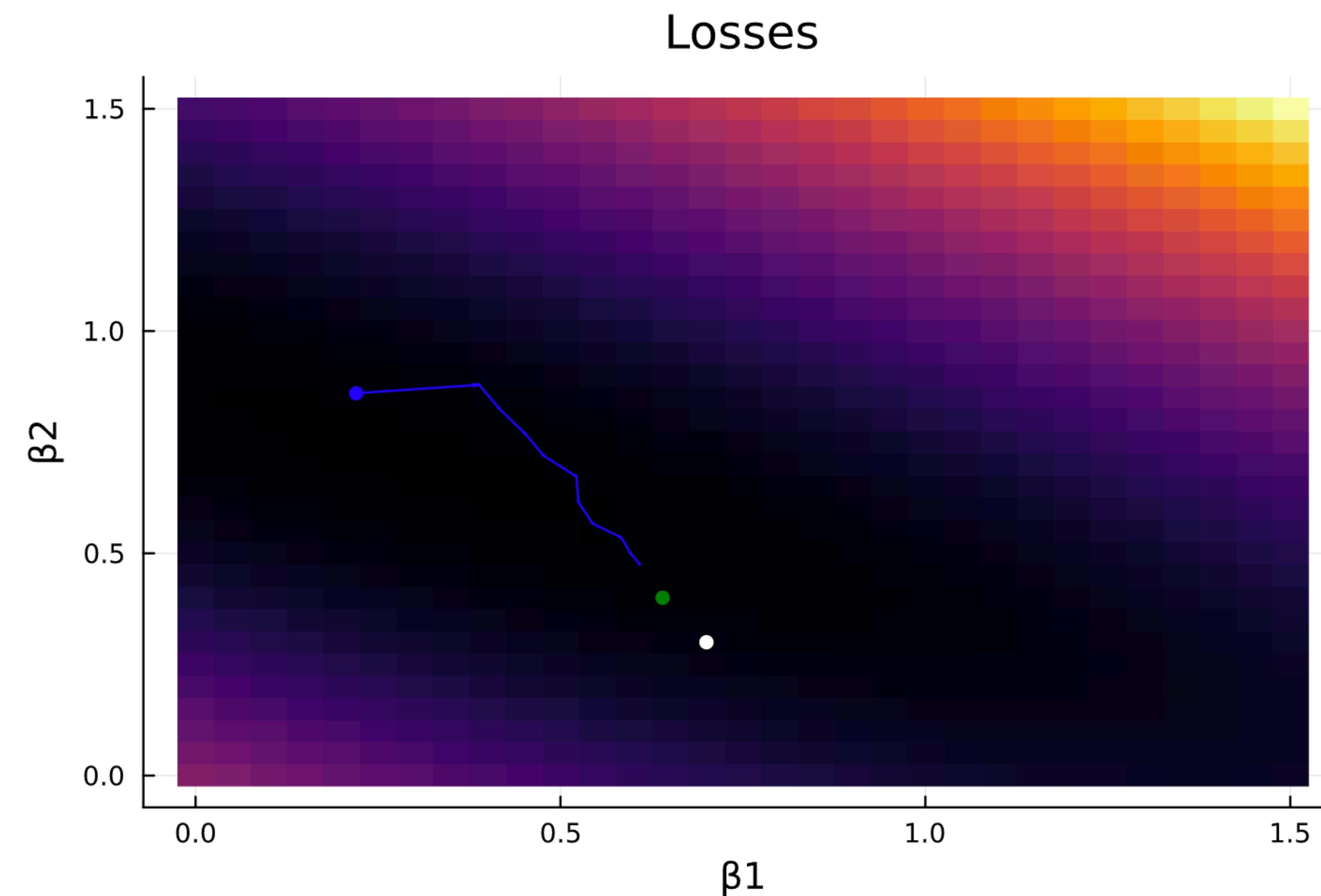
Stochastic Gradient Descent



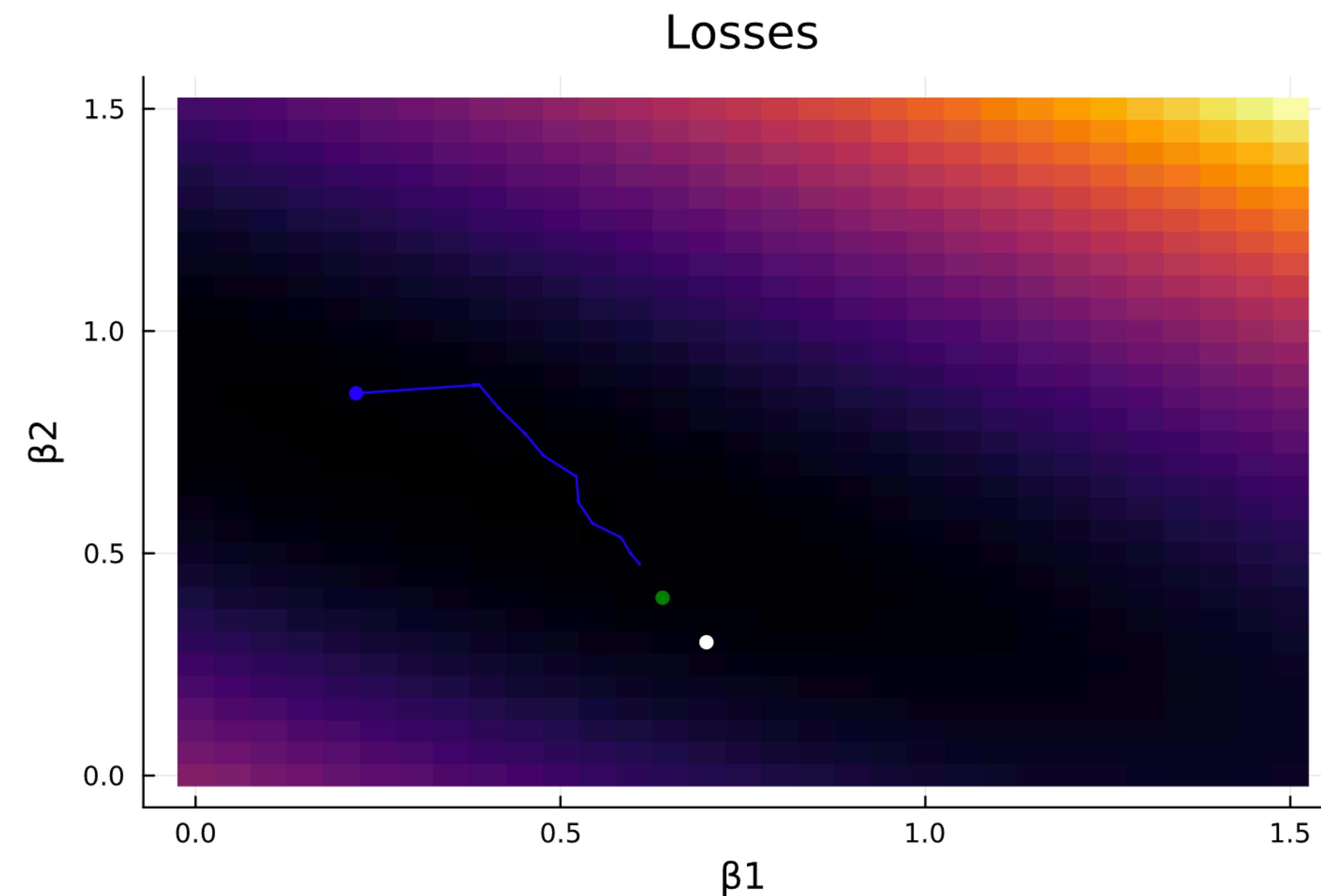
Stochastic Gradient Descent



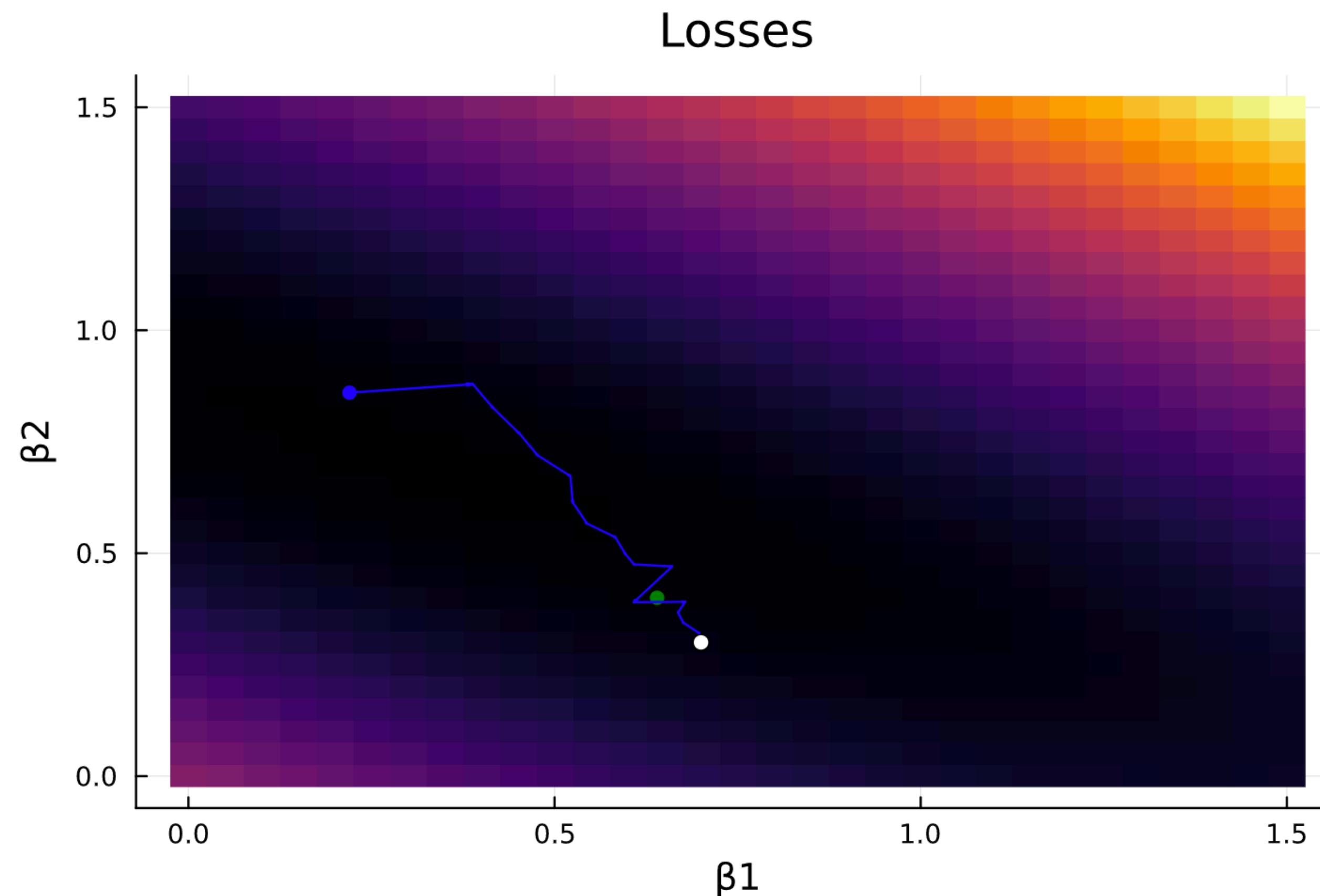
Stochastic Gradient Descent



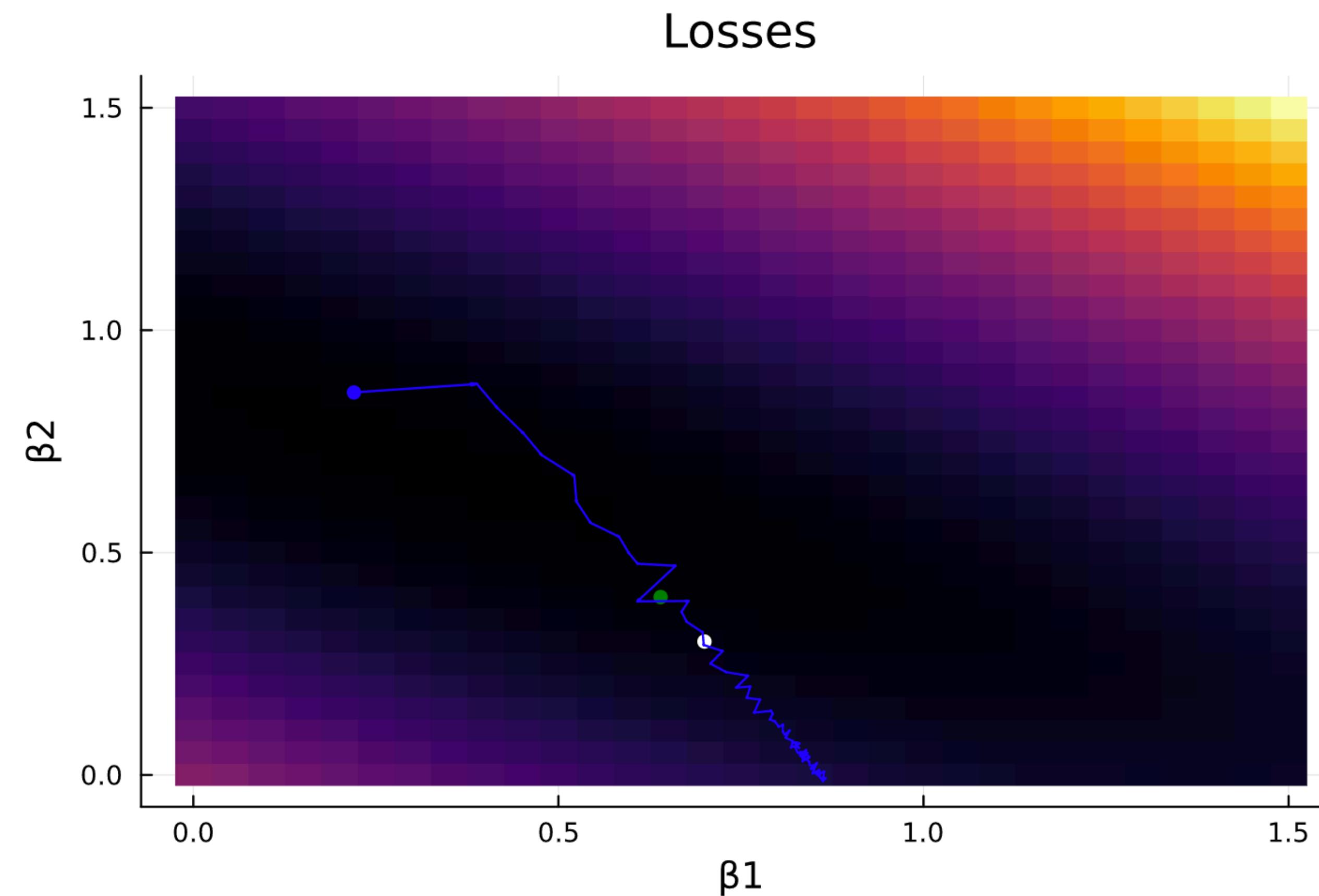
Stochastic Gradient Descent



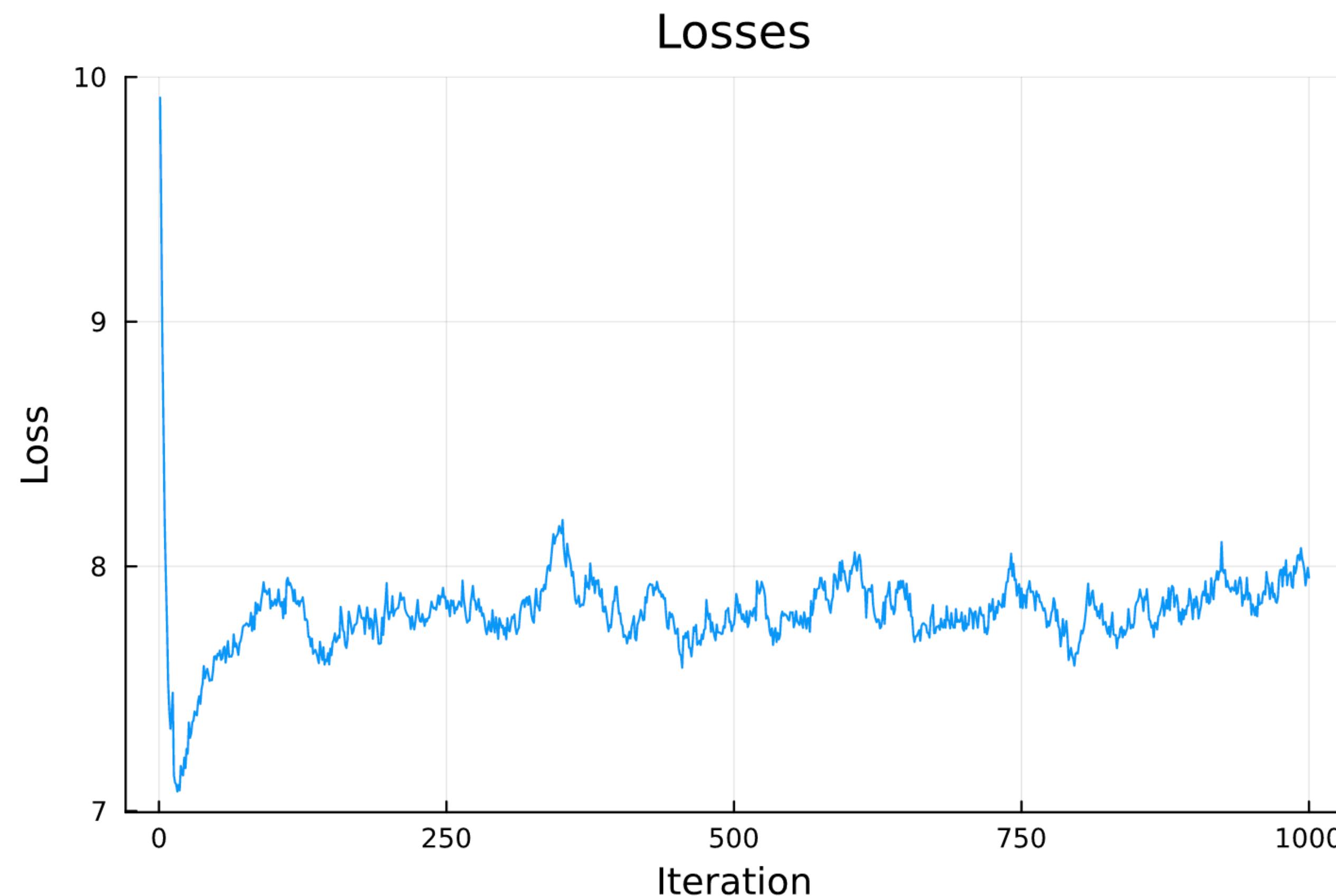
Stochastic Gradient Descent



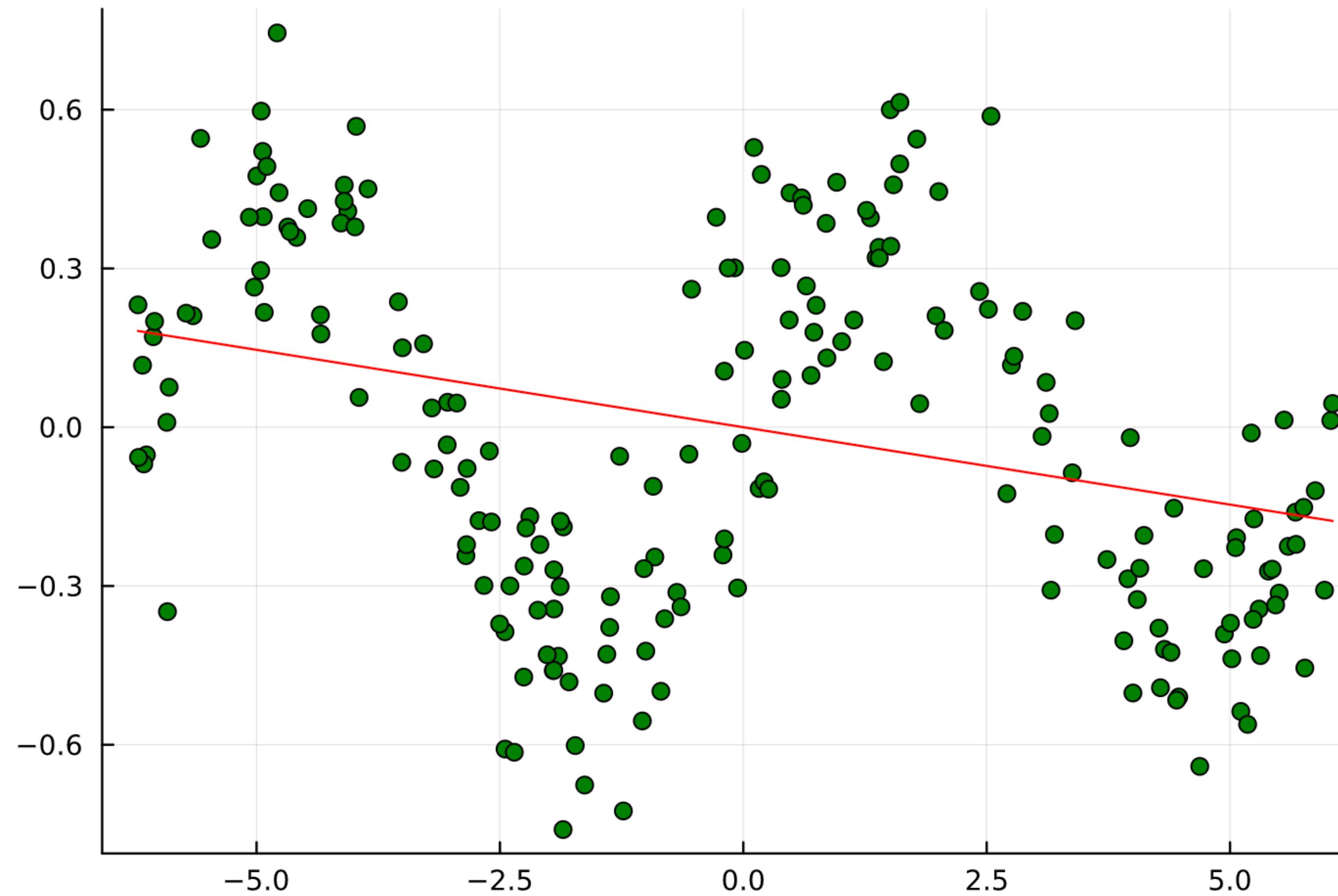
Stochastic Gradient Descent



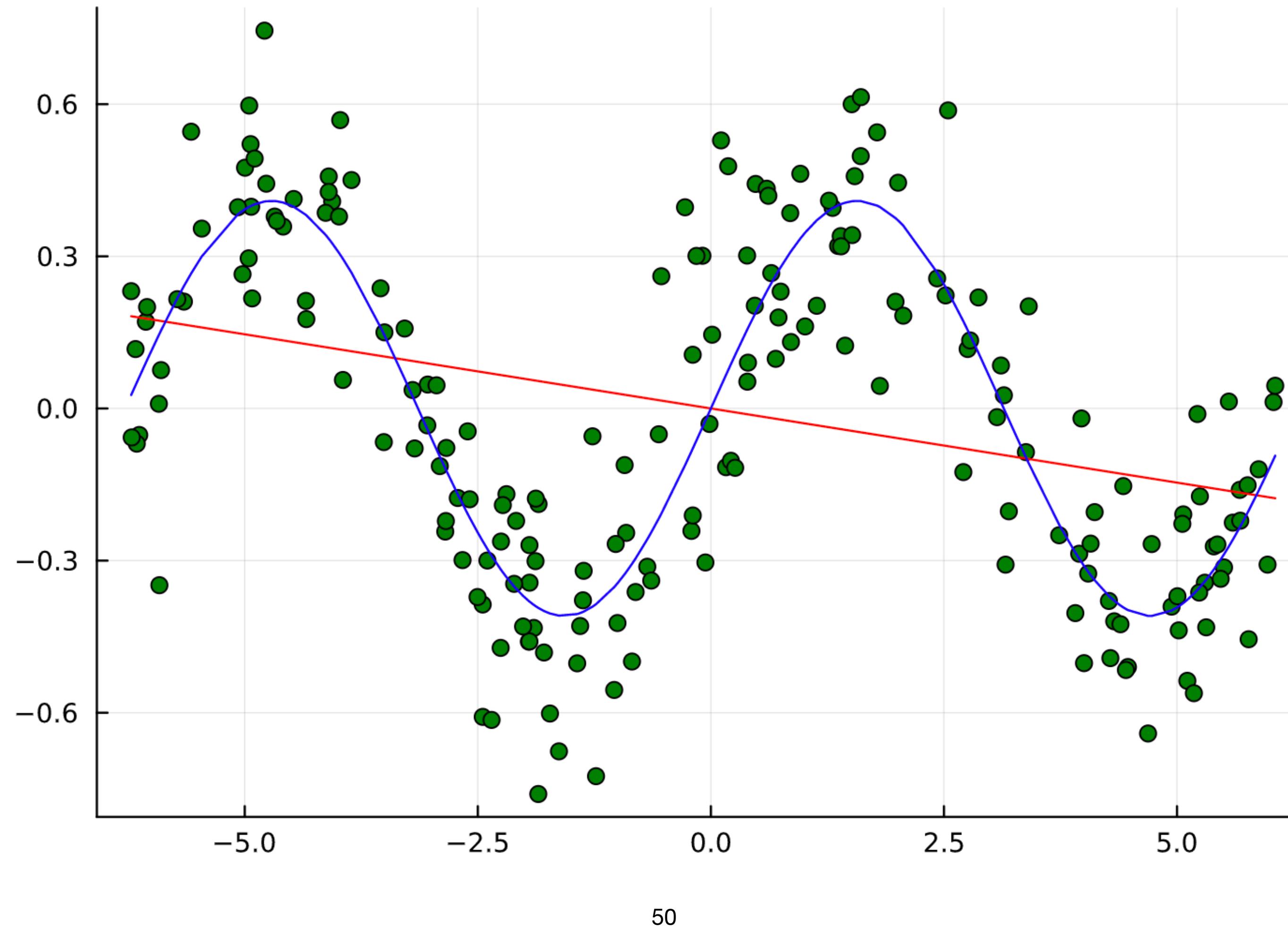
Stochastic Gradient Descent



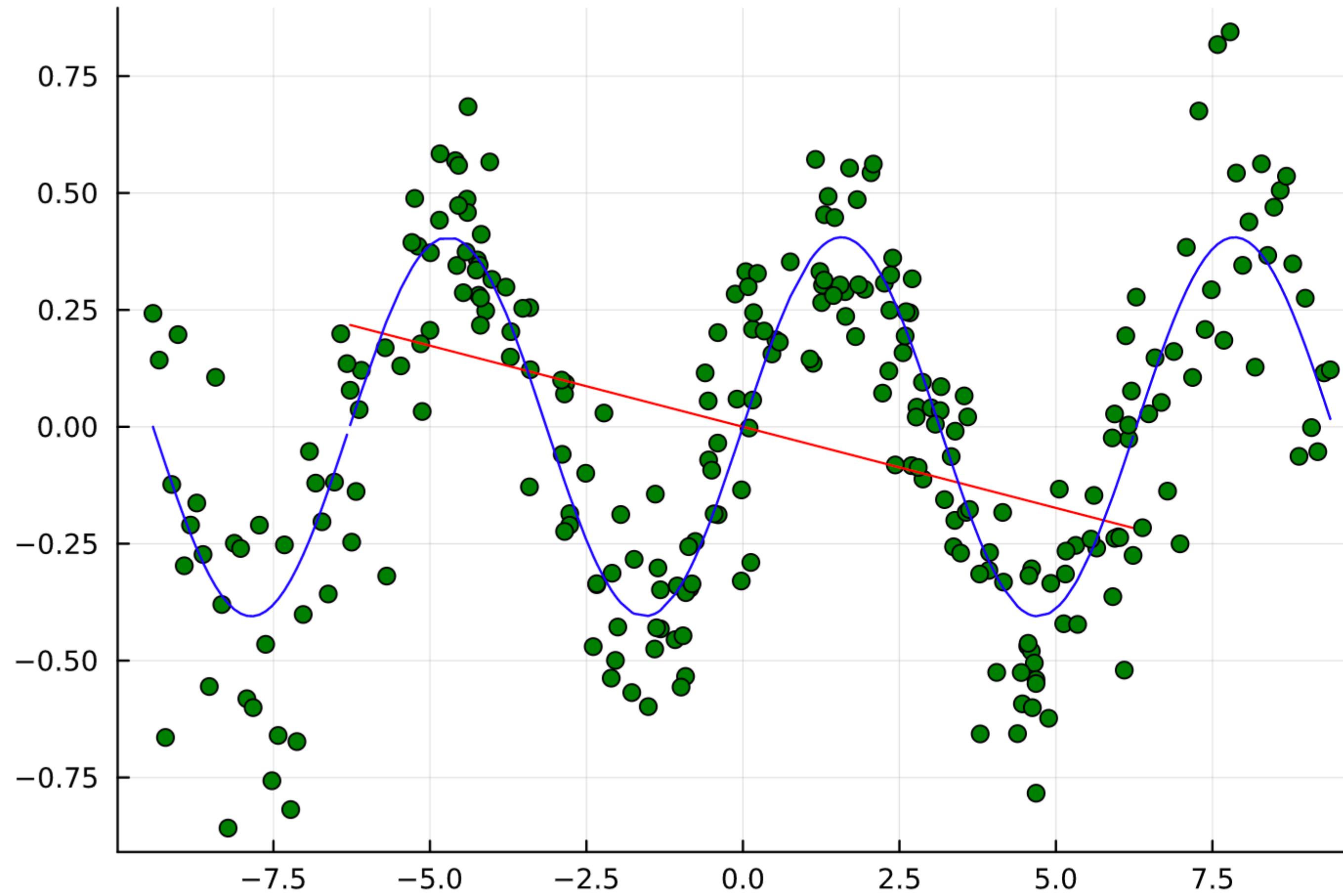
Feature Engineering



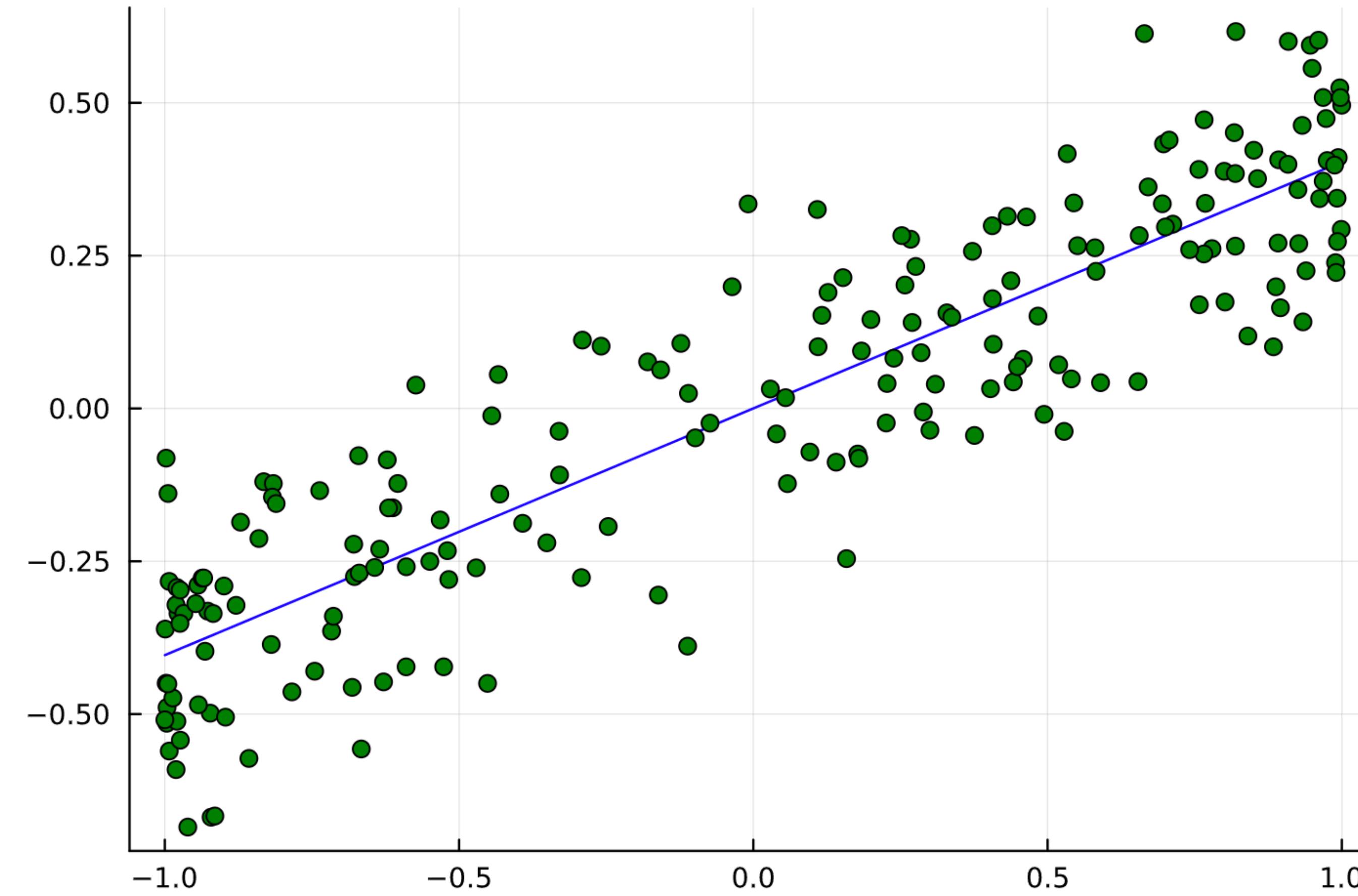
Feature Engineering



Feature Engineering

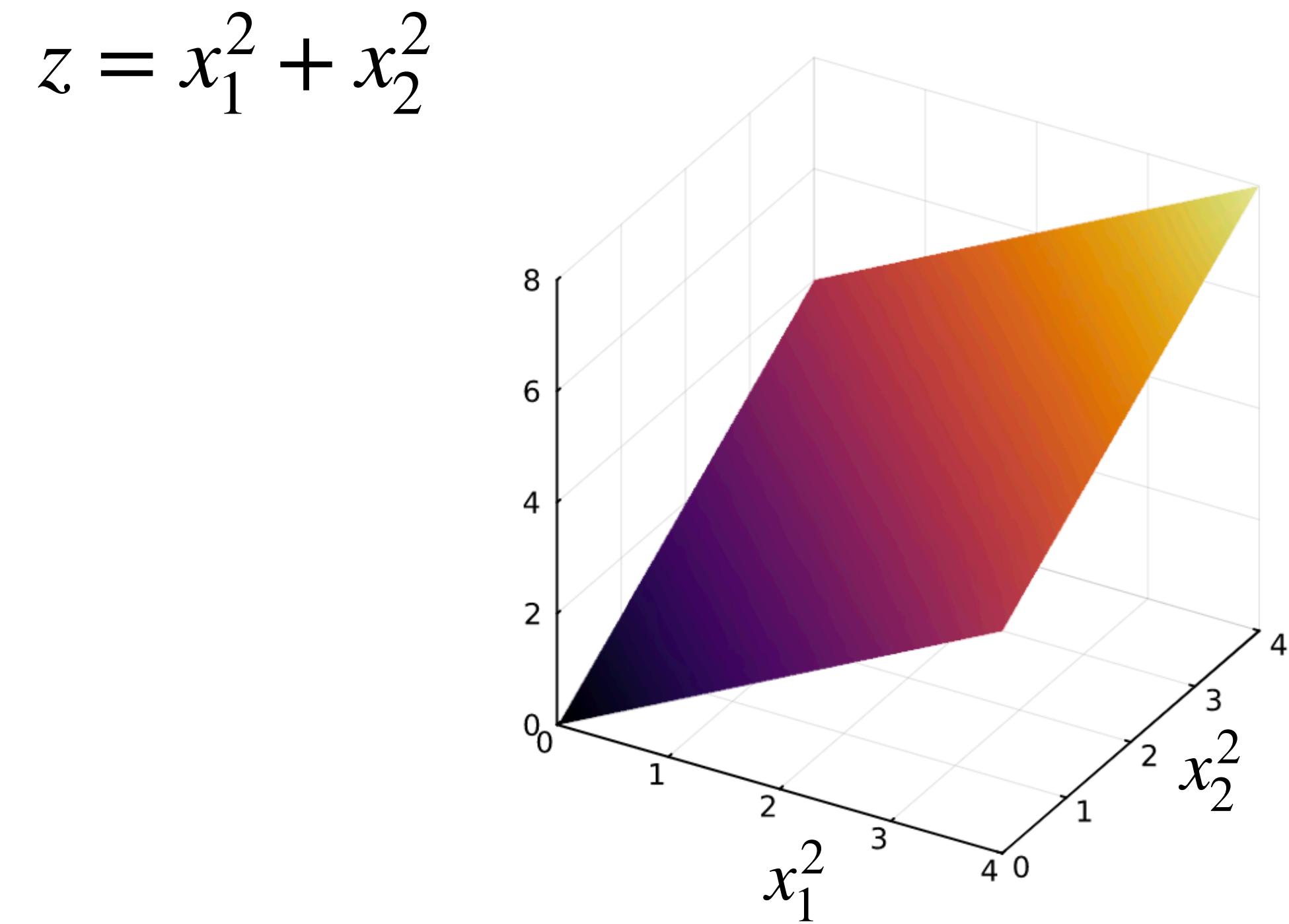
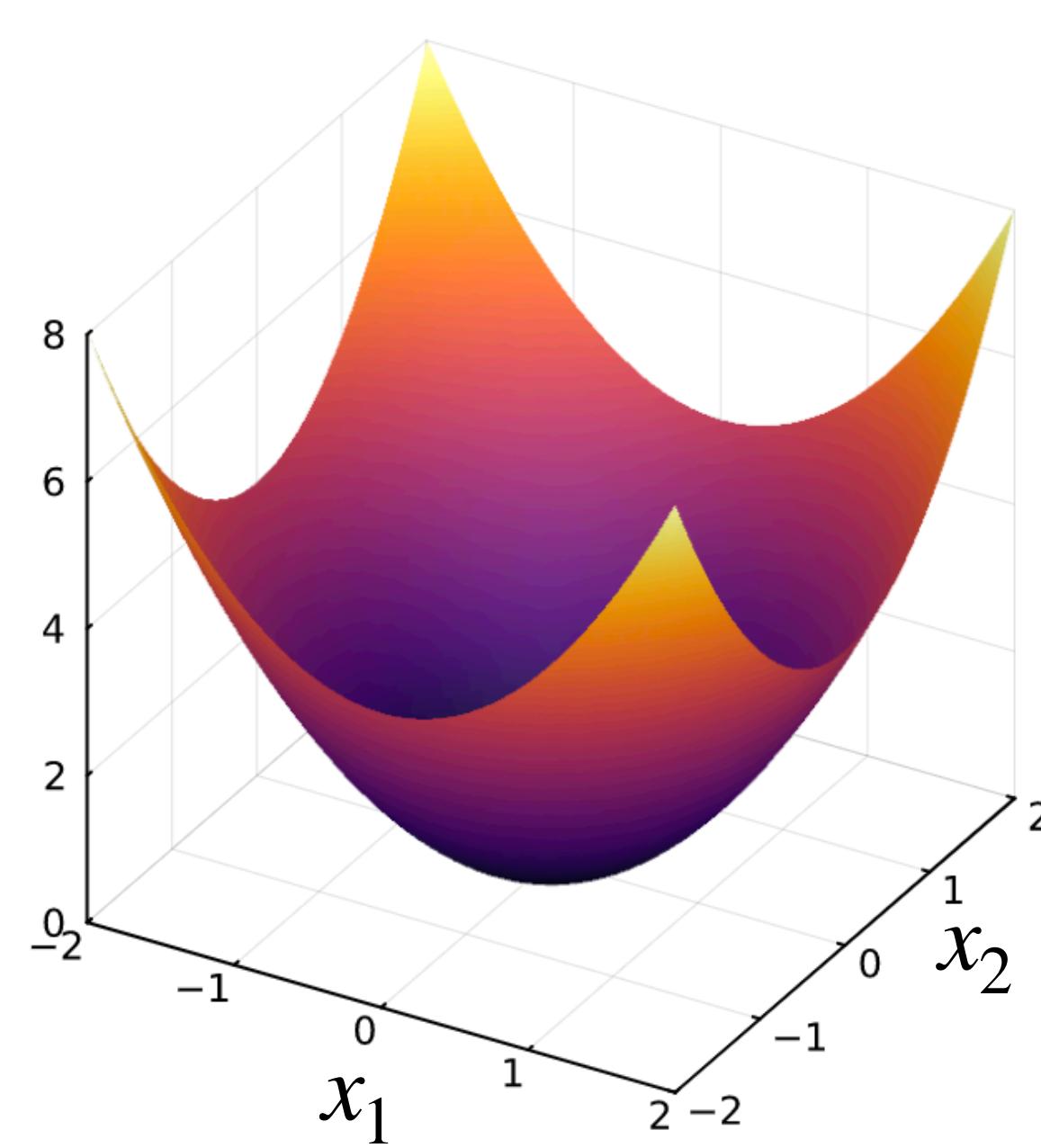


Feature Engineering

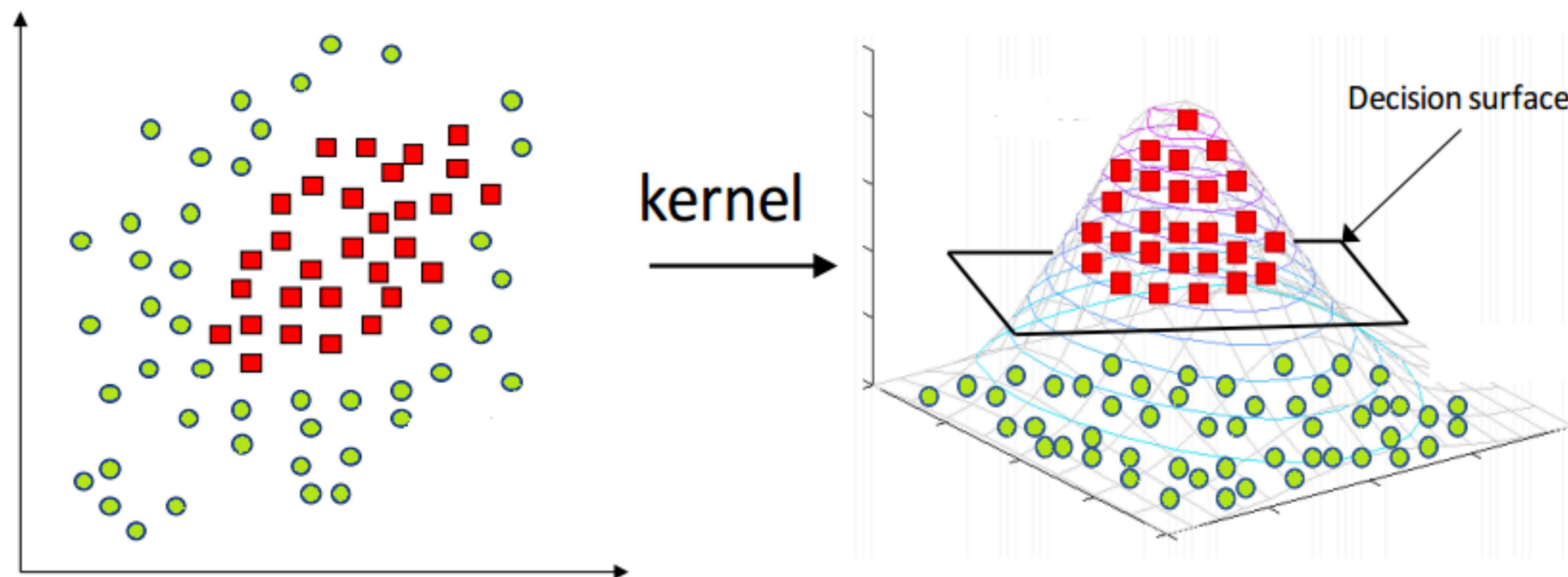


Kernel Trick

- Kernels are maps of non-linear data into a higher-dimensional space where the data becomes linearly separable



Kernel



Credit: Grace Zhang

Feature Engineering

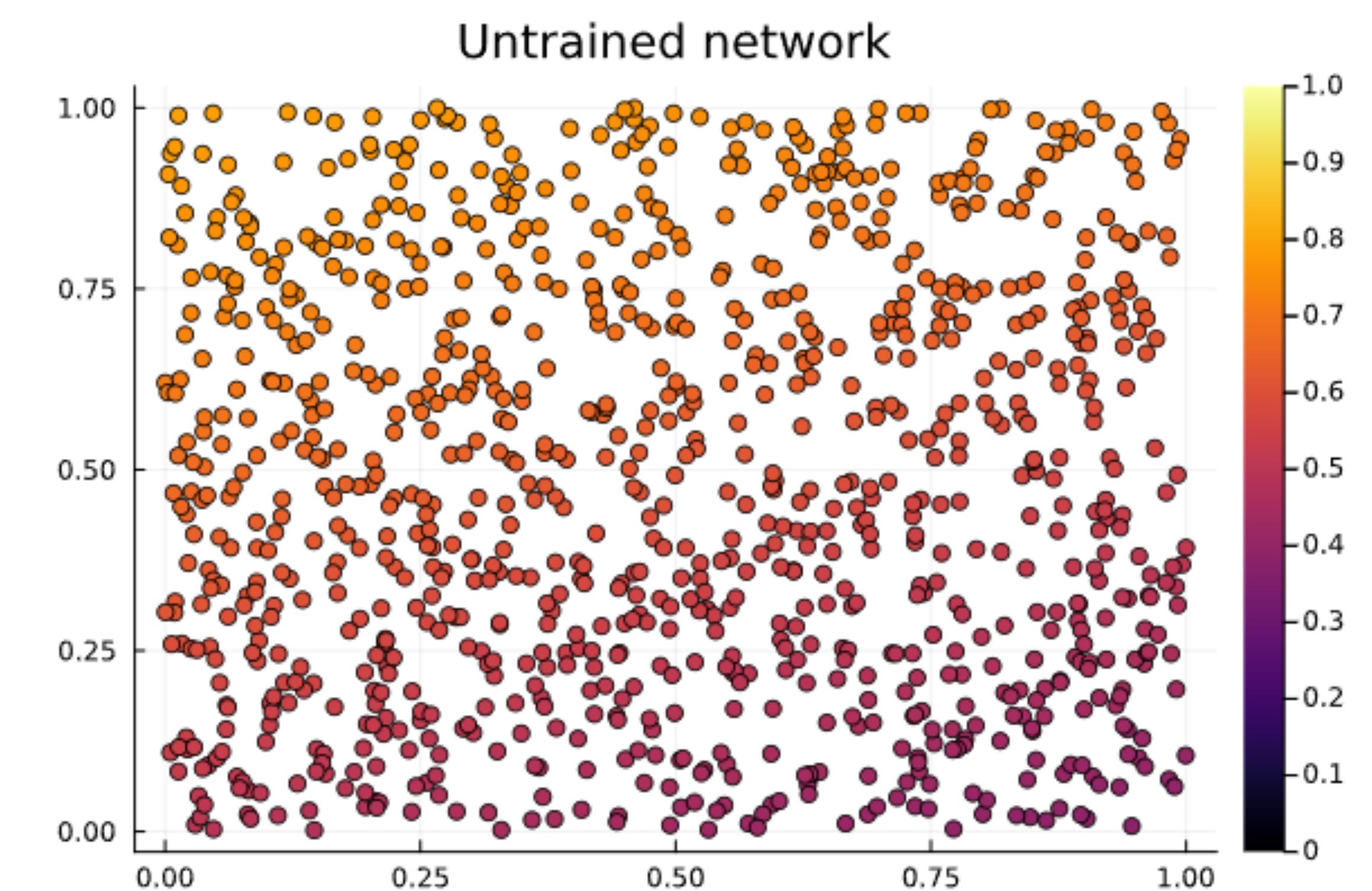
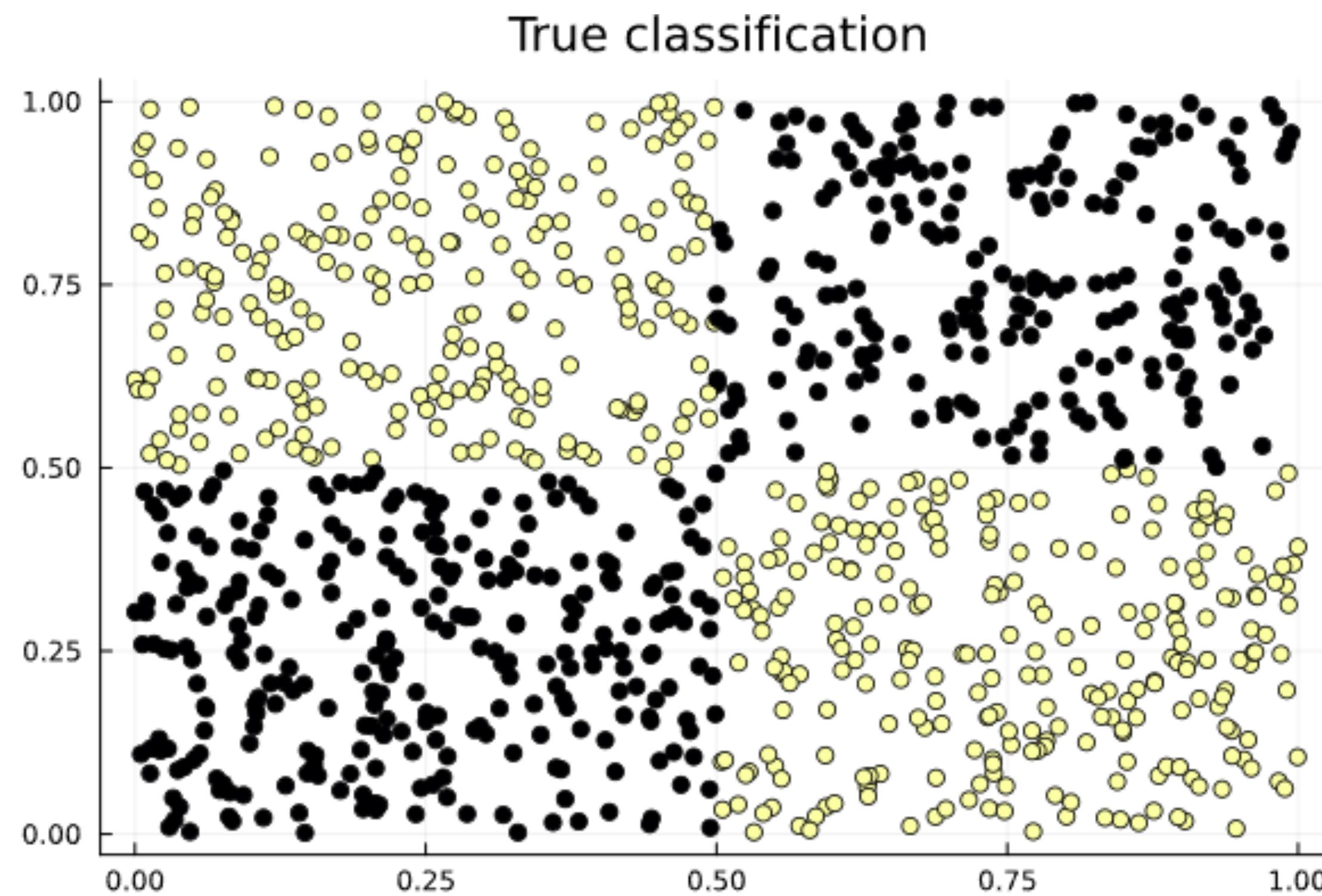
Neural networks can automatically learn relationships

$p = 2$ (features), $K = 2$ (layers), $g(z) = z^2$

$$W_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, b_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, W_2 = [1/4 \ - 1/4], b_2 = 0$$

$$\hat{f}(x) = \frac{1}{4} [1 \ - 1] \begin{bmatrix} (x_1 + x_2)^2 \\ (x_1 - x_2)^2 \end{bmatrix} = x_1 \cdot x_2$$

Classification



Classification

Implementation

```
# from https://fluxml.ai/Flux.jl/stable/models/quickstart/
using Flux, Statistics

# Generate some data for the XOR problem: vectors of length 2,
# as columns of a matrix:
noisy = rand(Float32, 2, 1000)
truth = [xor(col[1]>0.5, col[2]>0.5) for col in eachcol(noisy)]

# Define our model, a multi-layer perceptron with one hidden layer of size 3:
model = Chain(
    Dense(2 => 3, tanh), # activation function inside layer
    BatchNorm(3),
    Dense(3 => 2),
    softmax
) |> gpu # move model to GPU, if available
```

Classification

Implementation

```
# The model encapsulates parameters, randomly initialised.
```

```
# Its initial output is:
```

```
out1 = model(noisy |> gpu) |> cpu
```

```
# To train the model, we use batches of 64 samples, and one-hot encoding:
```

```
target = Flux.onehotbatch(truth, [true, false])
```

```
loader = Flux.DataLoader((noisy, target) |> gpu, batchsize=64, shuffle=true);
```

```
# will store optimiser momentum, etc.
```

```
optim = Flux.setup(Flux.Adam(0.01), model)
```

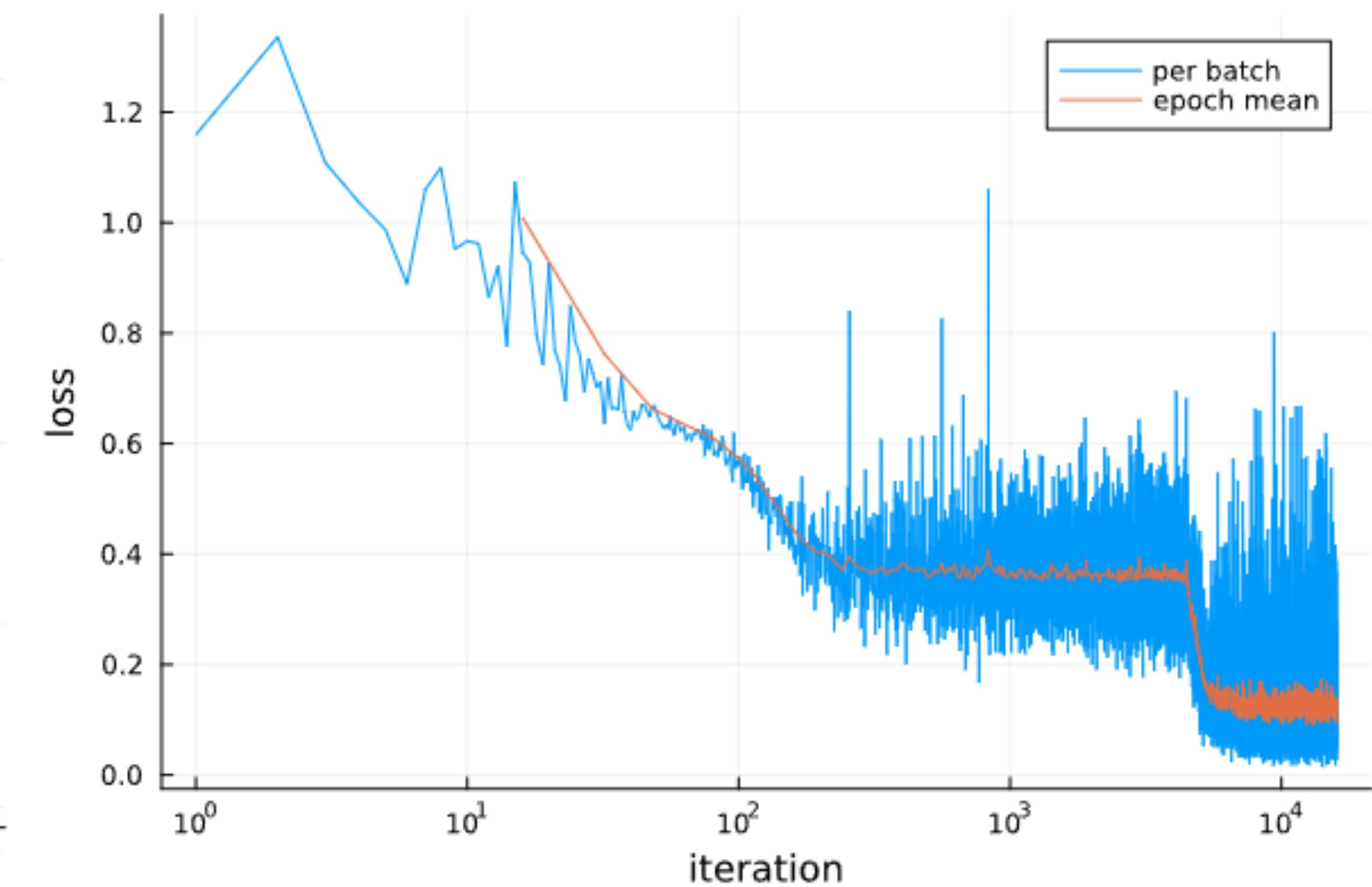
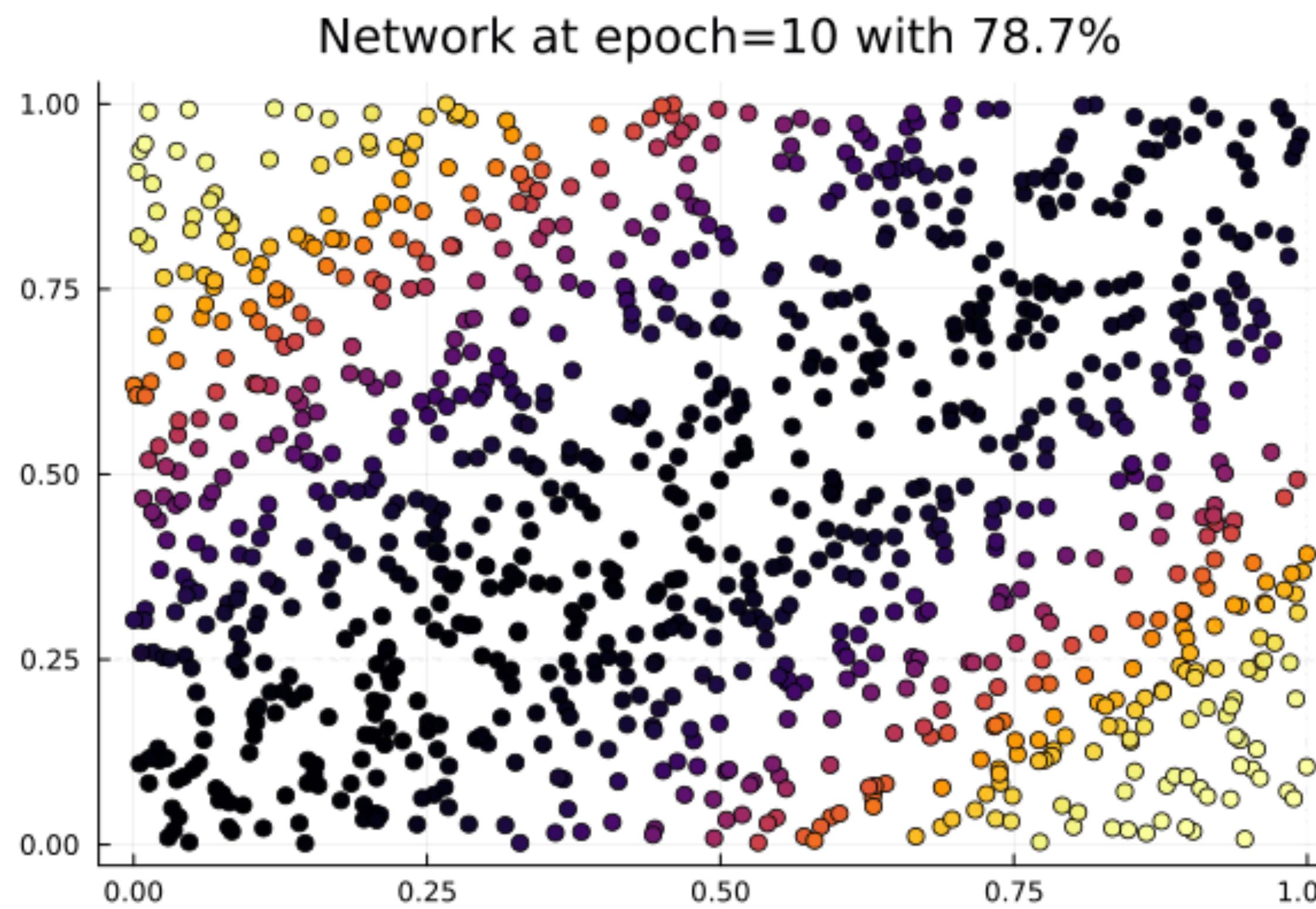
Classification

Implementation

```
# Training loop, using the whole data set 1000 times:  
losses = []  
for epoch in 1:1_000  
    for (x, y) in loader  
        loss, grads = Flux.withgradient(model) do m  
            # Evaluate model and loss inside gradient context:  
            y_hat = m(x)  
            Flux.crossentropy(y_hat, y)  
        end  
        Flux.update!(optim, model, grads[1])  
        push!(losses, loss) # logging, outside gradient context  
    end  
end
```

$$\begin{aligned} H(p, q) &= - \sum_{\forall x} p(x) \log(q(x)) \\ &= \sum_{\forall x} (p(x) \log(q(x)) + (1 - p(x)) \log(1 - q(x))) \end{aligned}$$

Classification



(água)



But unironically

GPT-4 visual input example, Moar Layers:

User Can you explain why this is funny. Think about it step-by-step.



GPT-4

The comic is satirizing the difference in approaches to improving model performance between statistical learning and neural networks.

In statistical learning, the character is shown to be concerned with overfitting and suggests a series of complex and technical solutions, such as minimizing structural risk, reworking the loss function, and using a soft margin.

In contrast, the neural networks character simply suggests adding more layers to the model. This is often seen as a common solution to improving performance in neural networks, but it's also considered a simplistic and brute-force approach.

The humor comes from the contrast between the complexity and specificity of the statistical learning approach and the simplicity and generality of the neural network approach. The "But unironically" comment adds to the humor by implying that, despite being simplistic, the "stack more layers" approach is often effective in practice.

Table 19: Example prompt demonstrating GPT-4's visual input capability.

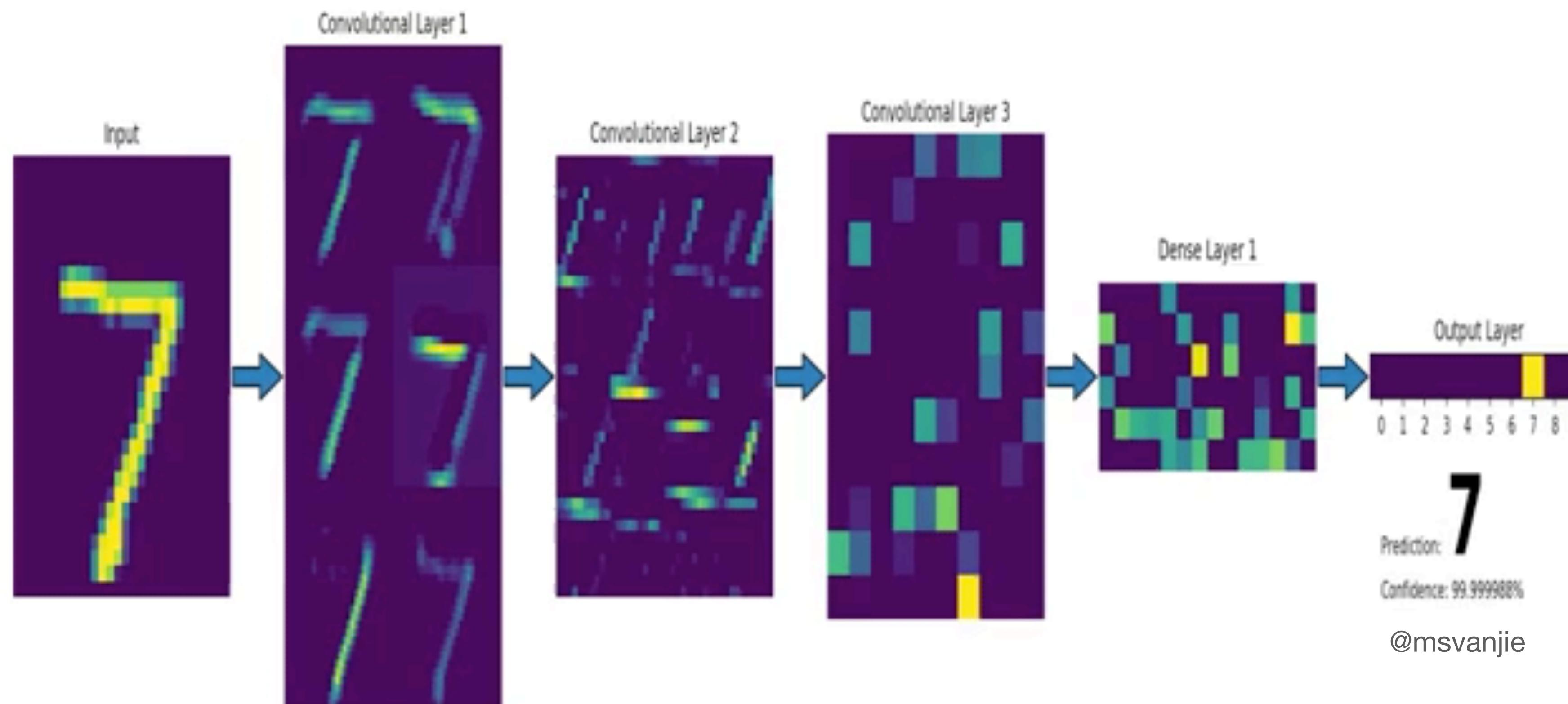
Convolutional Neural Networks

using Flux

```
# 28x28x1 -> 16x16x16 -> 3x3x32 -> 288 -> 10
model = Chain(
    Conv((3, 3), 1 => 16, stride = (1, 1), pad = (1, 1), relu),
    MaxPool((2, 2)),
    Conv((3, 3), 1 => 16, stride = (1, 1), pad = (1, 1), relu),
    MaxPool((2, 2)),
    Conv((3, 3), 1 => 16, stride = (1, 1), pad = (1, 1), relu),
    MaxPool((2, 2)),
    flatten, Dense(288, 10), softmax
) |> gpu
```

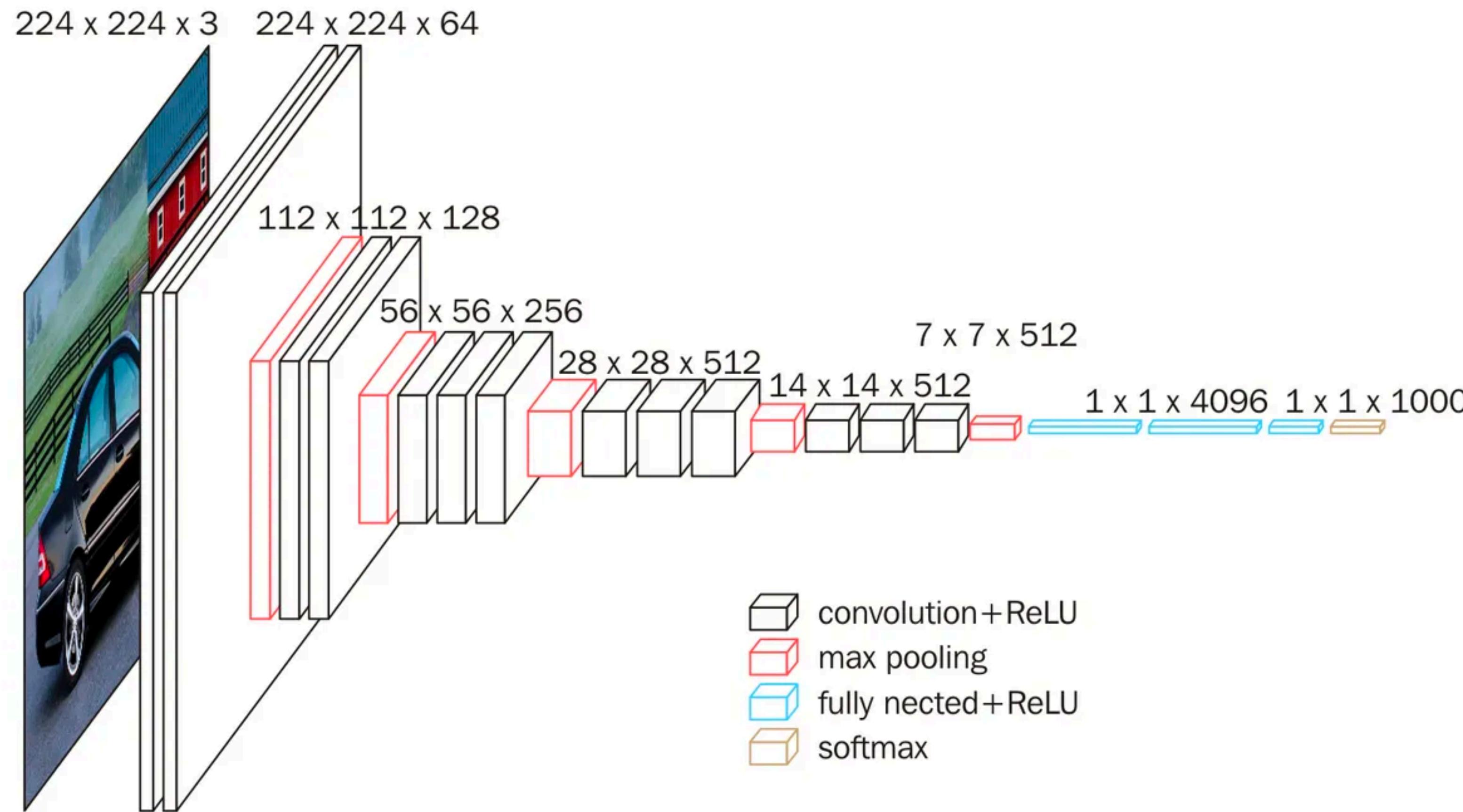
$$f(x_i) = \frac{e^{x_i}}{\sum_i e^{x_i}}$$

Convolutional Neural Networks



Foundational models

VGG16

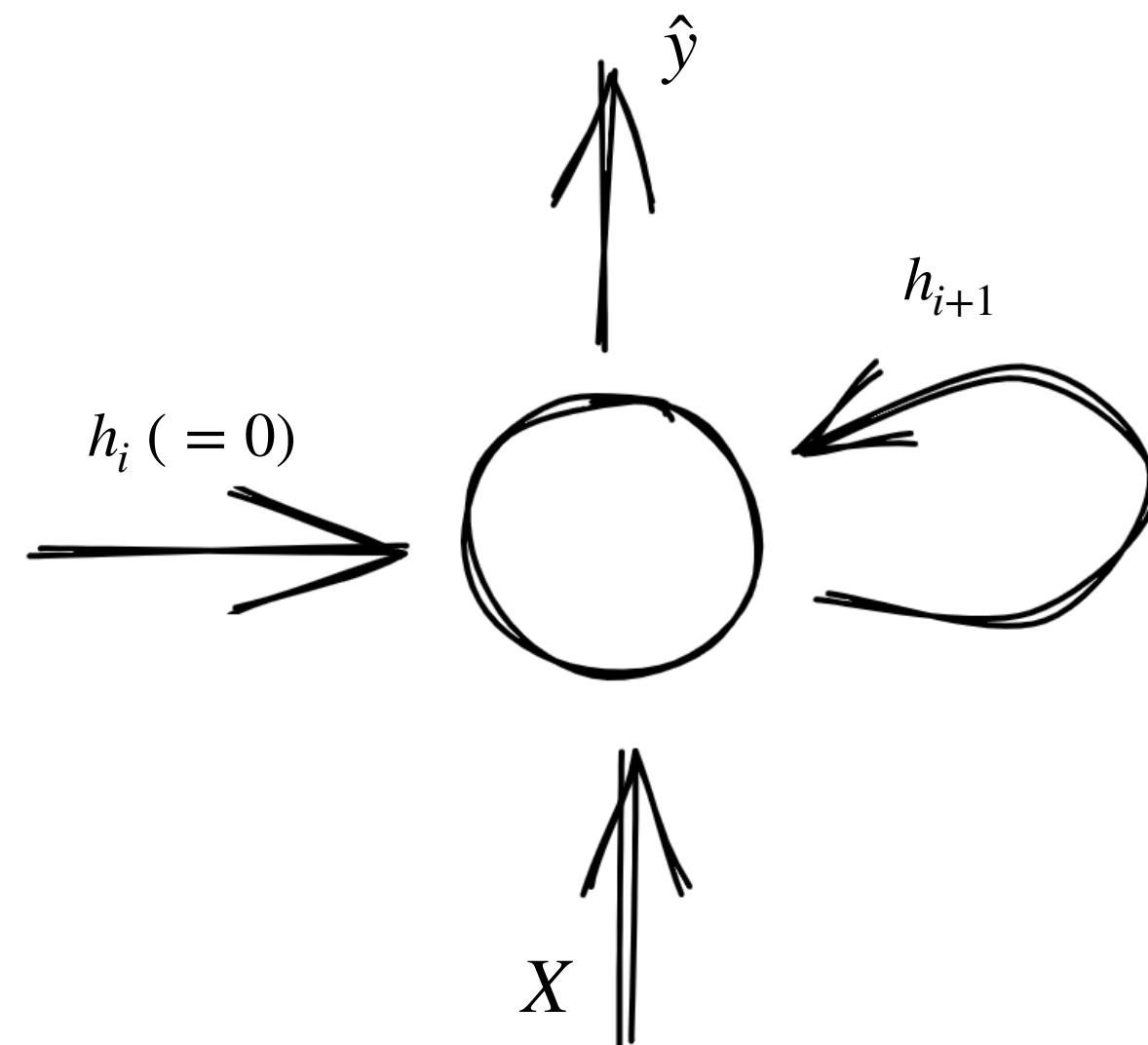


Recurrent Neural Networks

using Flux

```
# ... read and treat data ...
```

```
function eval_model(x)
    out = model.(x)[end]
    Flux.reset!(model)
    out
end
```

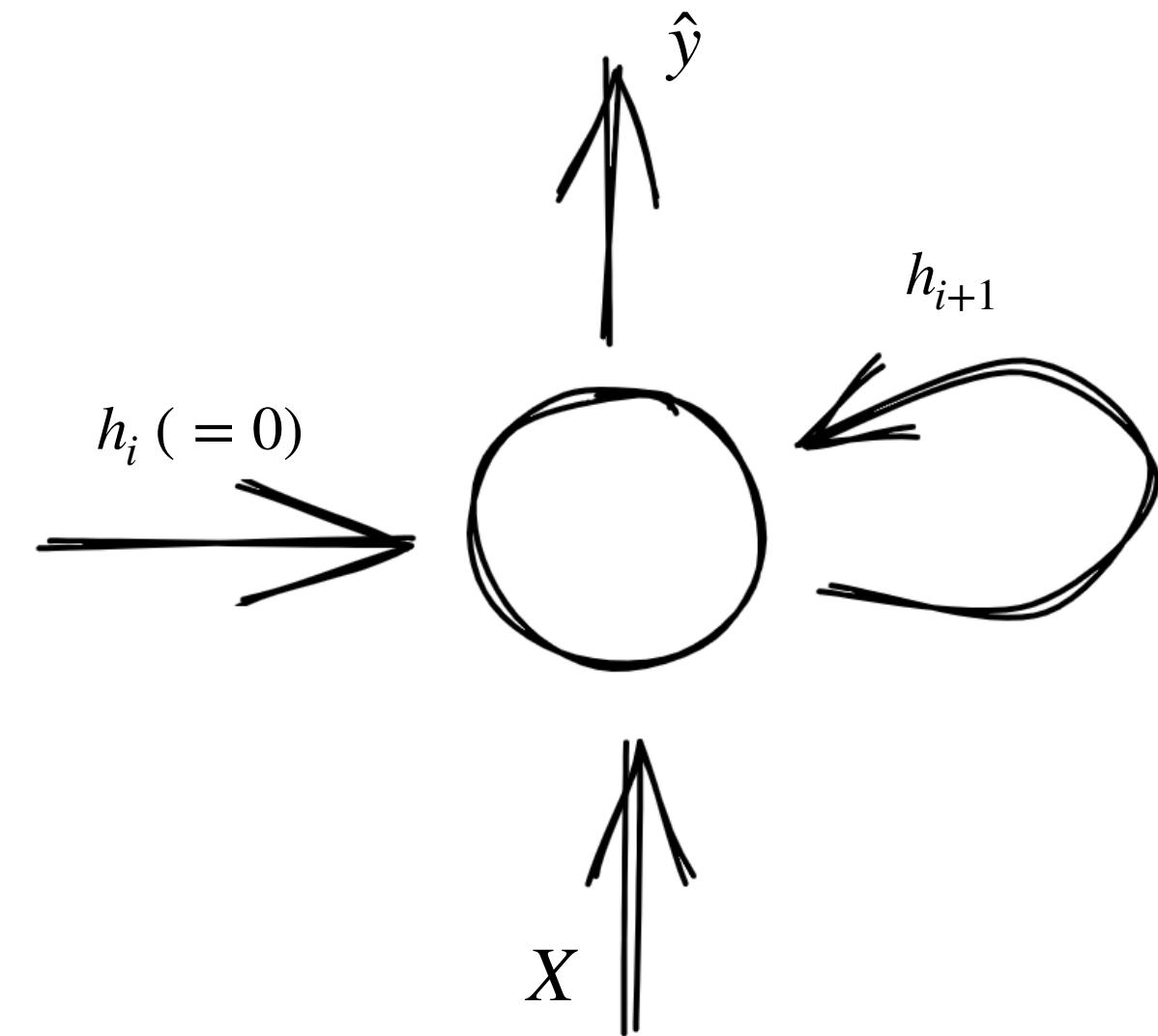


Recurrent Neural Networks

```
model = Chain(  
    RNN(1, 4, tanh),  
    Dense(4, 1)  
)
```

```
loss(x, y) = sum(abs.(eval_model(x) .- y)))  
ps = Flux.params(model)  
opt = Flux.ADAM()
```

```
@epochs num_epochs Flux.train!(  
    loss, ps, zip(train_data, train_labels), opt,  
    cb = Flux.throttle(evalcb, 1)  
)
```



<https://learningjulia.com/2019/10/11/a-basic-rnn.html#generate-data>

<https://fluxml.ai/Flux.jl/stable/training/callbacks/>

Feed Forward Networks and Optimal Trees

Figure 12.2: The first split of decision tree \mathcal{T}_1 .

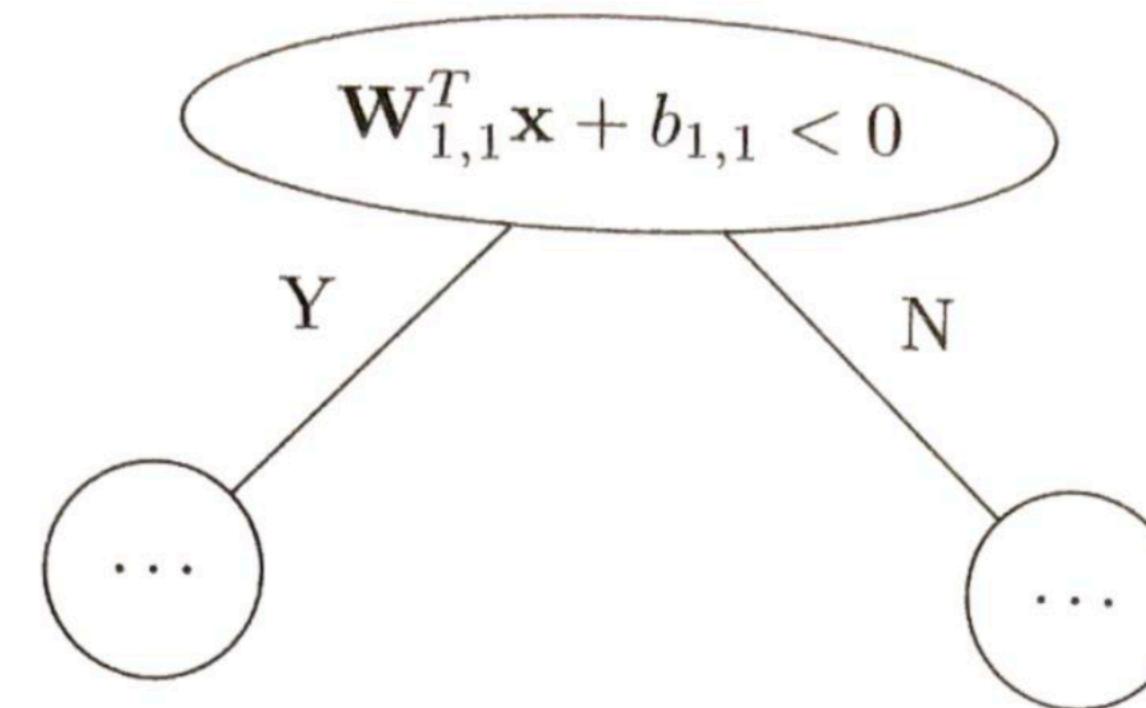
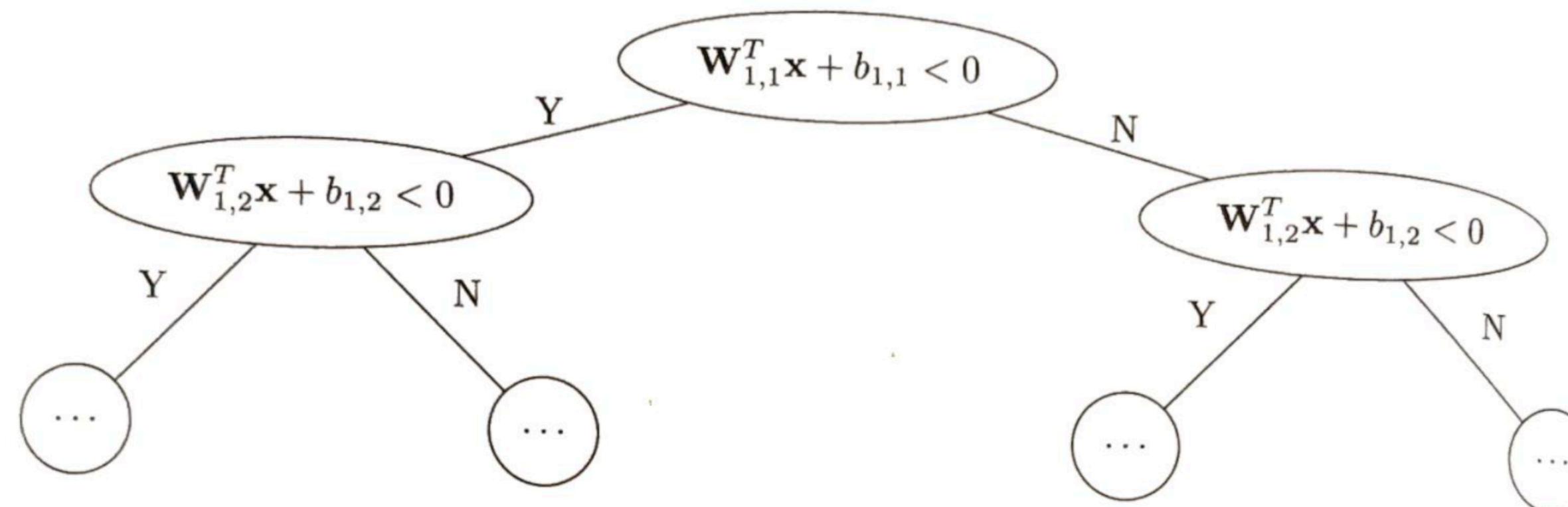


Figure 12.3: The first two depths of tree \mathcal{T}_1 .



Holistic Deep Learning

- Neural Networks are not robust: small changes in input can have drastic effects on outputs, rendering them vulnerable
- Different starting points can result in drastically different models and performances, reducing trust
- Neural Networks are not sparse, depending on a lot of non-zero parameters, limiting the hardware that can run them

Holistic Deep Learning

- Holistic Deep Learning is a framework to jointly optimize robustness (adversarial accuracy), stability (worst accuracy across train-val splits) and sparsity (parameters with value zero); showing that it is possible to improve all axis without sacrificing model performance
- Bertsimas et al propose a change on the loss function to include robustness, stability and sparsity

References

- Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani (2013). **An Introduction to Statistical Learning: with Applications in R**. New York. Springer.
- Dimitris Bertsimas, Kimberly Villalobos Carballo, Léonard Boussioux, Michael Lingzhi Li, Alex Paskov, Ivan Paskov. **Holistic Deep Learning**, 2021. <<http://arxiv.org/abs/2110.15829>'>arXiv:2110.15829>
- Michael Innes, Elliot Saba, Keno Fischer, Dhairyा Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, Viral Shah: **Fashionable Modelling with Flux**, 2018. <<http://arxiv.org/abs/1811.01457>>arXiv:1811.01457>

Extras

- Fischetti, M., Jo, J. **Deep neural networks and mixed integer linear optimization.** Constraints 23, 296–309 (2018). <<https://doi.org/10.1007/s10601-018-9285-6>>
- Nielsen, M. **A visual proof that neural nets can compute any function.** Neural Networks and Deep Learning (blog). <<http://neuralnetworksanddeeplearning.com/chap4.html>>
- shancarter, dsmilkov. **TensorFlow Playground.** <<https://playground.tensorflow.org/>>

Obrigado!