

# 參考資料

- [https://docs.juliahub.com/DiffEqFlux/BdO4p/1.13.0/examples/NeuralODE\\_Flux/](https://docs.juliahub.com/DiffEqFlux/BdO4p/1.13.0/examples/NeuralODE_Flux/)
- <https://turing.ml/dev/tutorials/03-bayesian-neural-network/>
- <https://turing.ml/dev/tutorials/12-gaussian-process/>
- <https://medium.com/analytics-vidhya/time-series-prediction-feat-introduction-of-julia-78ed6897910c>
- <https://iwasnothing.medium.com/use-transformer-in-julia-8409d2c0b642>
- <https://gist.github.com/luiarthur/7a1dfa6a980d11a00862152a371a5cf3>
- <https://juliagaussianprocesses.github.io/AbstractGPs.jl/dev/examples/2-deep-kernel-learning/>
- <https://github.com/tallamjr/tubingen-probabilistic-ml>
- <https://fluxml.ai/Flux.jl/stable/>
- <https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf>

# 課題概要

- ・データは以下のurlの気象庁の1日の気温データを使う
- ・<https://www.data.jma.go.jp/gmd/risk/obsdl/index.php>
- ・本課題について：目的：juliaを使って最低限の計算ができるようそのための練習(個人的に研究はやりたくない)と1行のデータだけ使ってこのデータの1日後の様子を予測できる方法を探しだす（もし可能であれば 練習のため 1行の気温データ（今日と明日の最高気温）を使う。
- ・期待できる効果：1行のデータだけで気温予測できるモデルの発見

# データについて

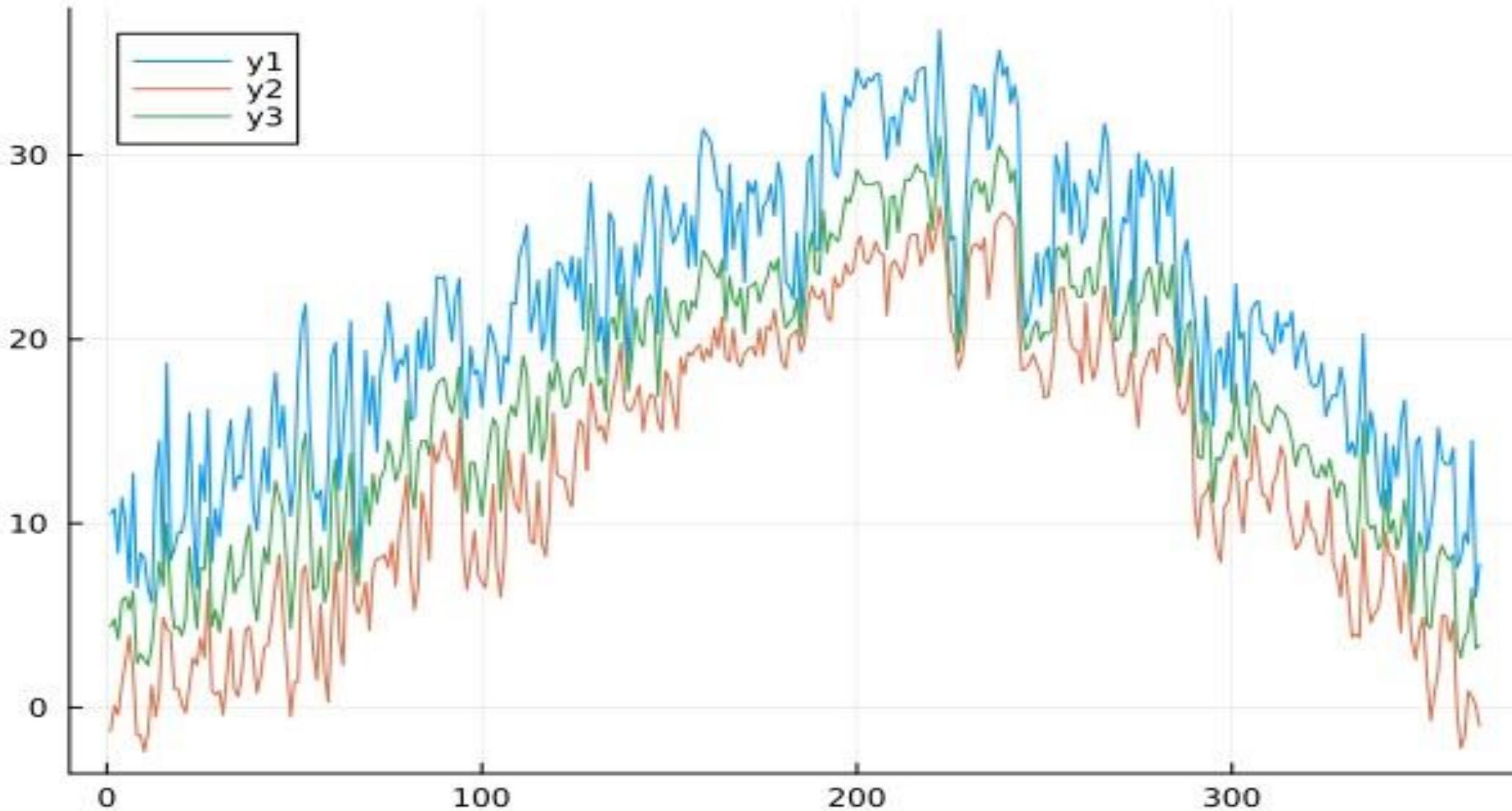
横軸が日（1 – 3 6 5）縦軸は気温として使う

この後は一般に最高気温を使う時々他のも使う 理由は実際可視化とコサイン類似度計算すると三つのデータ（最高気温、最低気温、平均気温はほぼ同じパターンなので(2021年のデータを例に最高気温と平均気温の類似度は約0.96) コサイン類似度を使う理由は $k \times T(a) = T(b)$   $\cos$ は一つのベクトルをもう一つのベクトルへの投影として認識できる 1 に近いほど同じ線になり線形的な関係を示す

# Julia で可視化してみる(2021年) コード

```
using CSV,DataFrame, Plots  
data=CSV.read("w2021.csv",DataFrame,header=6)  
• plot(data[1:end,5])  
• plot!(data[1:end,8])  
• plot!(data[1:end,2])
```

# 可視化結果



# 方法

- 使った手法を2種類を分ける
- 手法 1 : 微分方程式,rnn(lstm gru) transformer
- 手法 2 : 確率方法 (ベイズ、ガウス過程、カーネル法、ベイジアンニューラルネットワーク、deep kernel learning)
- 具体の流れ
- 微分方程式8~20
- ニューラルネットワーク21~43
- 確率方法44~63

# 方法1:微分方程式でシミュレーション（最適化つきで）

- まず2021年の365日の最高気温と最低気温を取る x軸は日数 y軸は温度

$$\frac{dT}{dt} = \frac{(1-a)s}{c} - \sigma T^4$$

$$C_a \frac{dT_a}{dt} = \varepsilon\sigma T_s^4 - 2\varepsilon\sigma T_a^4$$

$$C_s \frac{dT_s}{dt} = \varepsilon\sigma T_a^4 - \sigma T_s^4 + (1 - \alpha)Q(\phi)$$

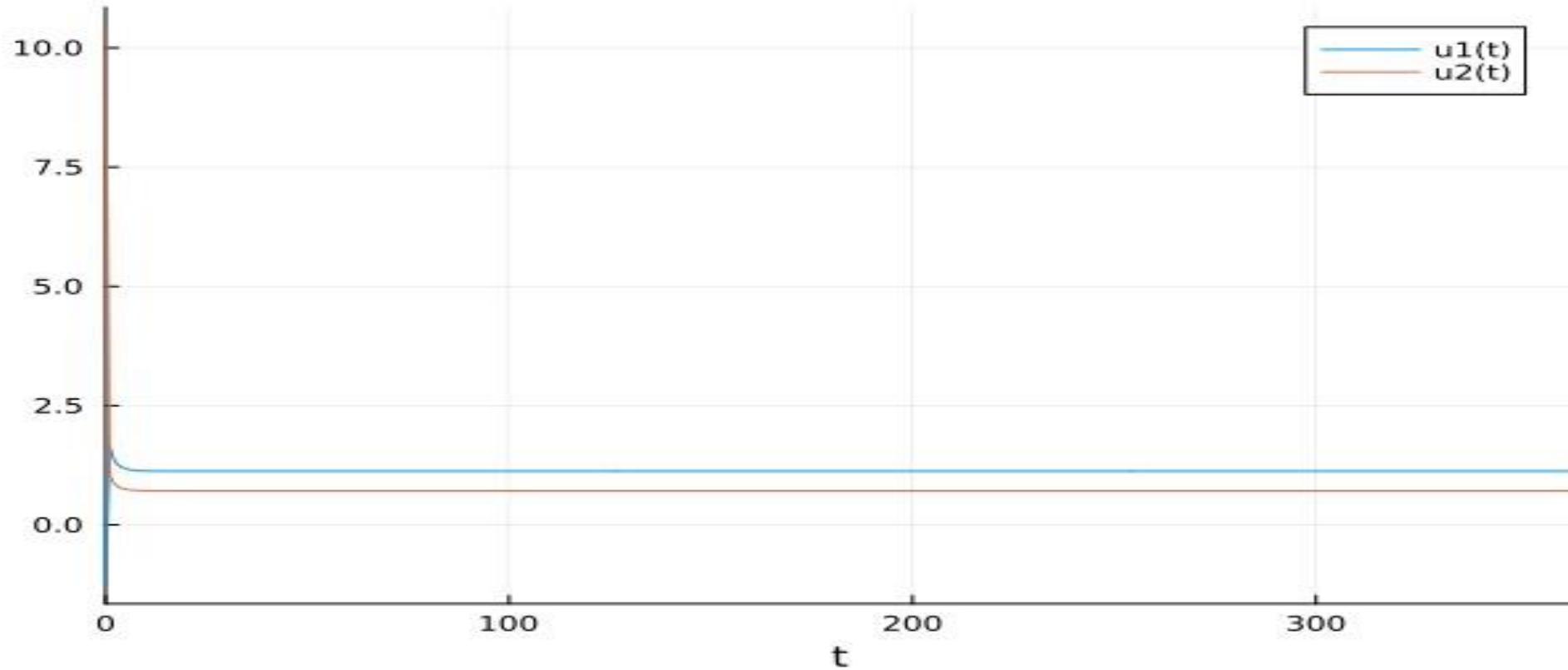
- 微分方程式はシュテファン=ボルツマンの法則と半分の地球の面積が受けた熱量を入れて作った式

- 実際計算する時t以外の数値を全部パラメーターにする
- ここで最高気温と最低気温二つの項目を入れて計算する
- Juliaのsciml用のライブラリを使って計算するまず微分方程式を初期化してシミュレーションする パラメータを(0,1)の範囲で初期化（こうするとエラーがでやすいが一応このままにする）
- この方法の計算速度はかなり遅い

# コード

- **using**  
DiffEqFlux,OrdinaryDiffEq,Flux,Optim,Plots,DifferentialEquations,CSV,DataFrames,ForwardDiff #ライブラリ
- **function** **tts**(du,u,p,t) #微分方程式
- ta,ts=u
- ca,cs,si,e,a,q=p
- du[1]=dta=(1/ca)\*(si\*ts^4 - 2\*si\*e\*ta^4)
- du[2]=dts=(1/cs)\*(si\*e\*ta^4 - si\*ts^4)+(1-a)\*q
- **end**
- u0=[-1.3,10.5] #初期条件
- tspan=(0,364)
- p=rand(6)
- prob=ODEProblem(tts,u0,tspan,p)
- sol=solve(ODEProblem(tts,u0,tspan,p),saveat=1)
- **plot**(sol) #プロット

# 結果を見る



# 最適化してみる

- 損失関数を定義して adamで最適化 DiffEqFluxライブラリはもし初期パラメータはランダムだとエラーが出やすい（試したコードは訓練回数2回しかできない）
- コード

- `function loss(p) #損失関数定義`

- `tmp_prob =remake(prob,p=p)`

- `tmp_sol=solve(tmp_prob,saveat=1)`

- `return sum(abs2,Array(tmp_sol)[:,:2:end]-dataset[:,:,:2:end])`

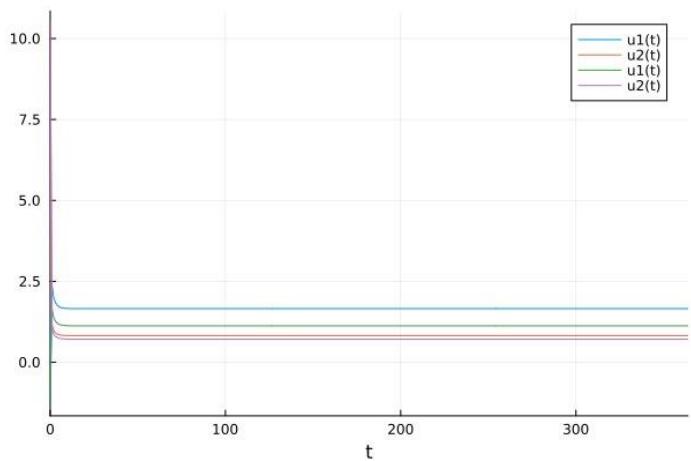
- `end`

- `res=DiffEqFlux.sciml_train(loss,p,Adam(0.05),maxiters=10) #最適化 得られるのは更新したパラメータ`

- `sol1=solve(ODEProblem(tts,u0,tspan,res),saveat=1) #もう一度シミュレーション`

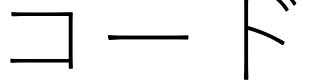
- `plot(sol1)`

# 結果を見る(失敗)



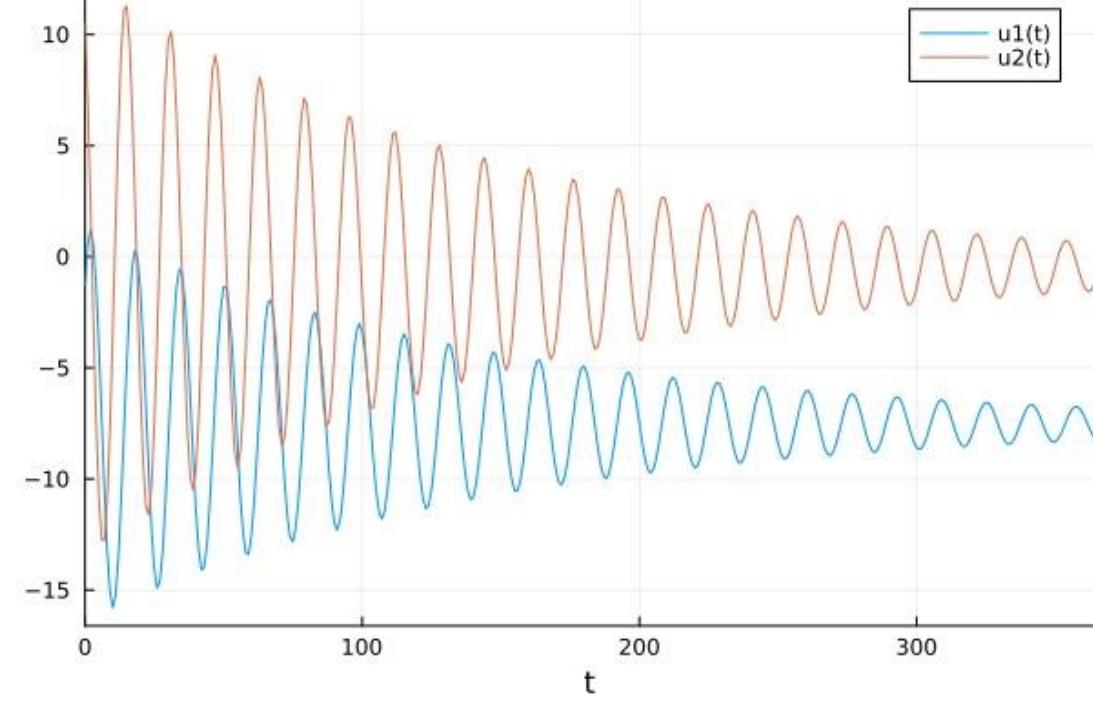
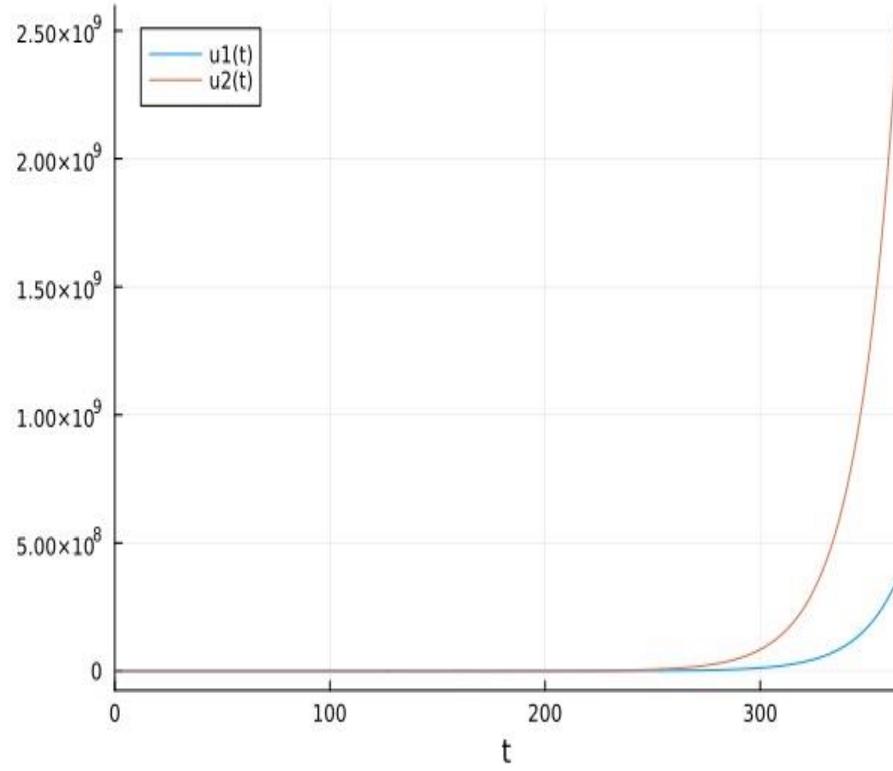
# ニューラルネットワークを取り入れる

- 少し前の方方法変える：ニューラルネットワークを入れる
- 理由：ニューラルネットワークは関数の近似に強い（mlpを例に）  
たとしたら微分方程式の近似も可能
- 具体的に：まずニューラルネットワーク定義その後  $\frac{dT}{dt} = \text{NN}(T)$  にする
- 今回は線形mlpとtanhと入力データを  $x^4$  にしたものを試す(これだけで定義すると結果はうまくいかない 実際ニューラルネットワーク以外のパラメータも必要がその方法がわからぬので 今回はこのままにする)



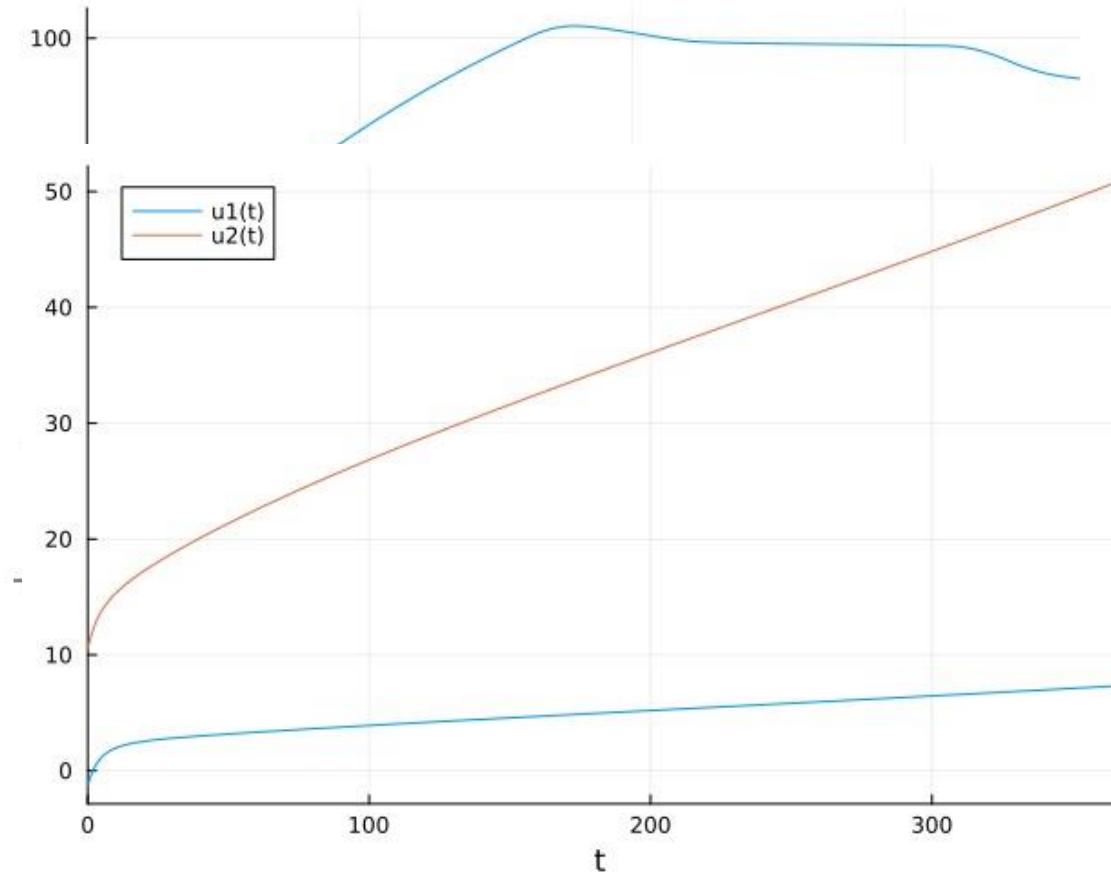
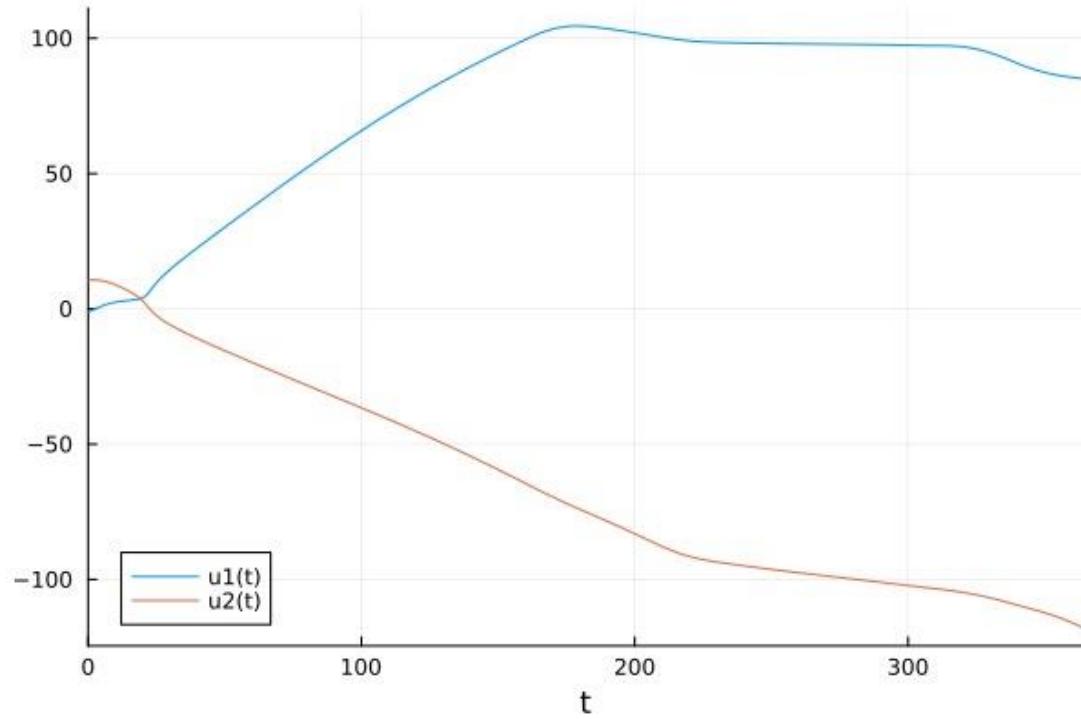
- `dudt2=FastChain((x,p)-> x,`
- `FastDense(2,50,),`
- `FastDense(50,10,),`
- `FastDense(10,2))`
- `neural_ode(u,p,t)=dudt2(u,p)`
- `p=initial_params(dudt2)`
- `prob=ODEProblem(neural_ode,u0,tspan,p)`
- `sol=solve(ODEProblem(neural_ode,u0,tspan,p),saveat=1)`
- `res=DiffEqFlux.sciml_train(loss,p,Adam(0.05),maxiters=100)`
- `sol1=solve(ODEProblem(neural_ode,u0,tspan,res),saveat=1)`

# 線形mlpシミュレーションと最適化結果

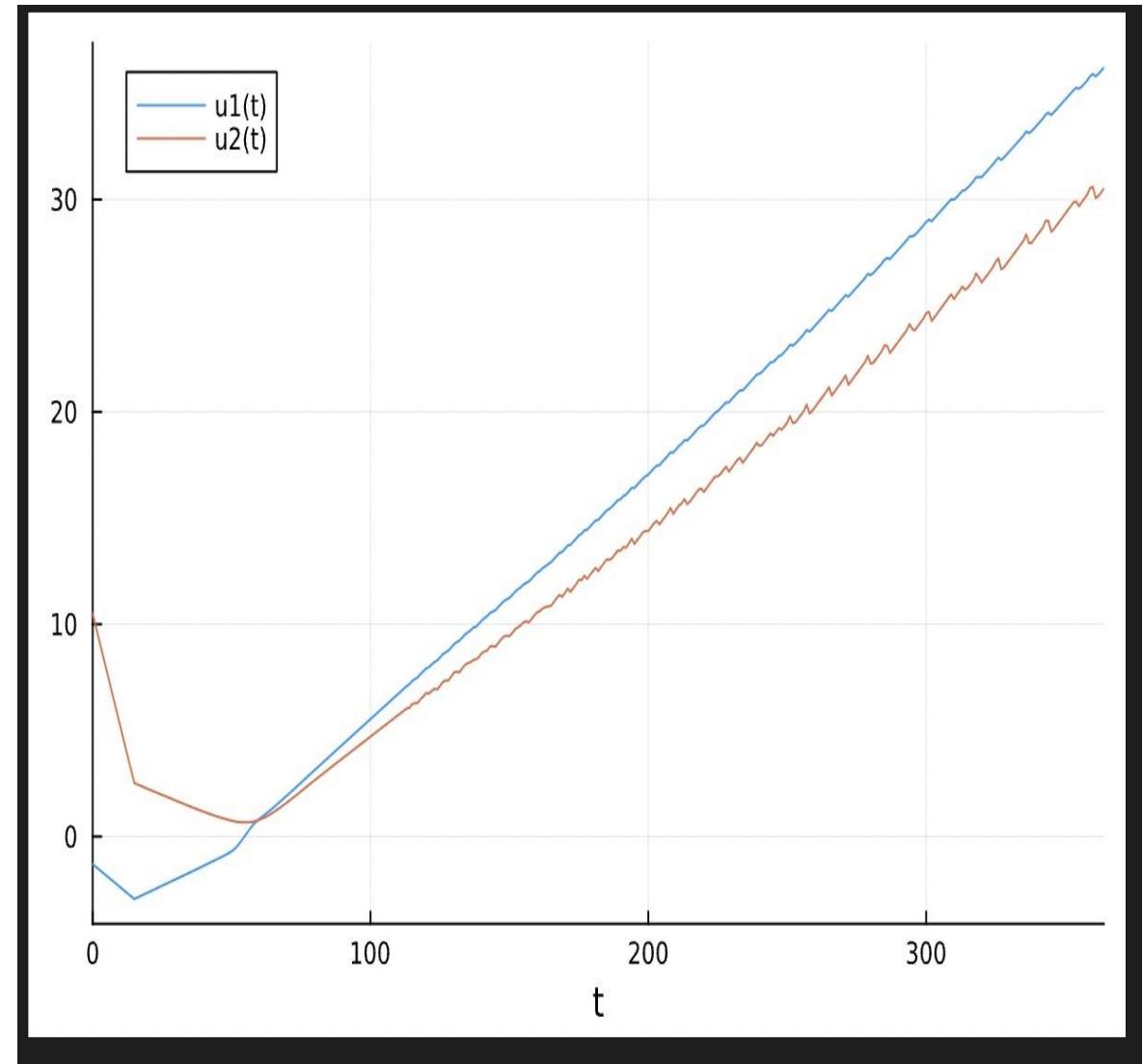
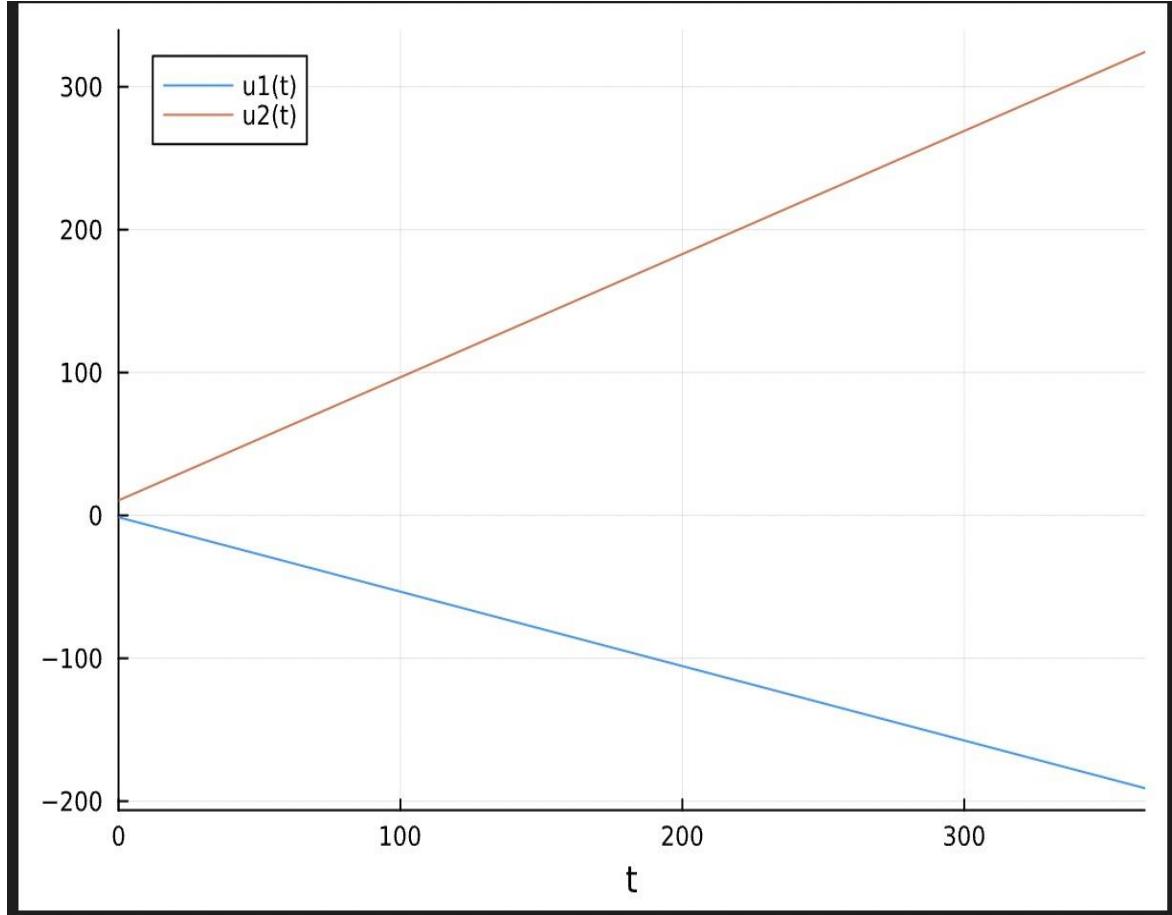


# 隠し層tanh 出力は線形

- `dudt2=FastChain((x,p)-> x, FastDense(2,50,tanh), FastDense(50,10,tanh), FastDense(10,2))`



入力を $x^4$ にする



# 2次微分方程式 $\frac{d^2y}{d^2x} = NN(x)$ にする

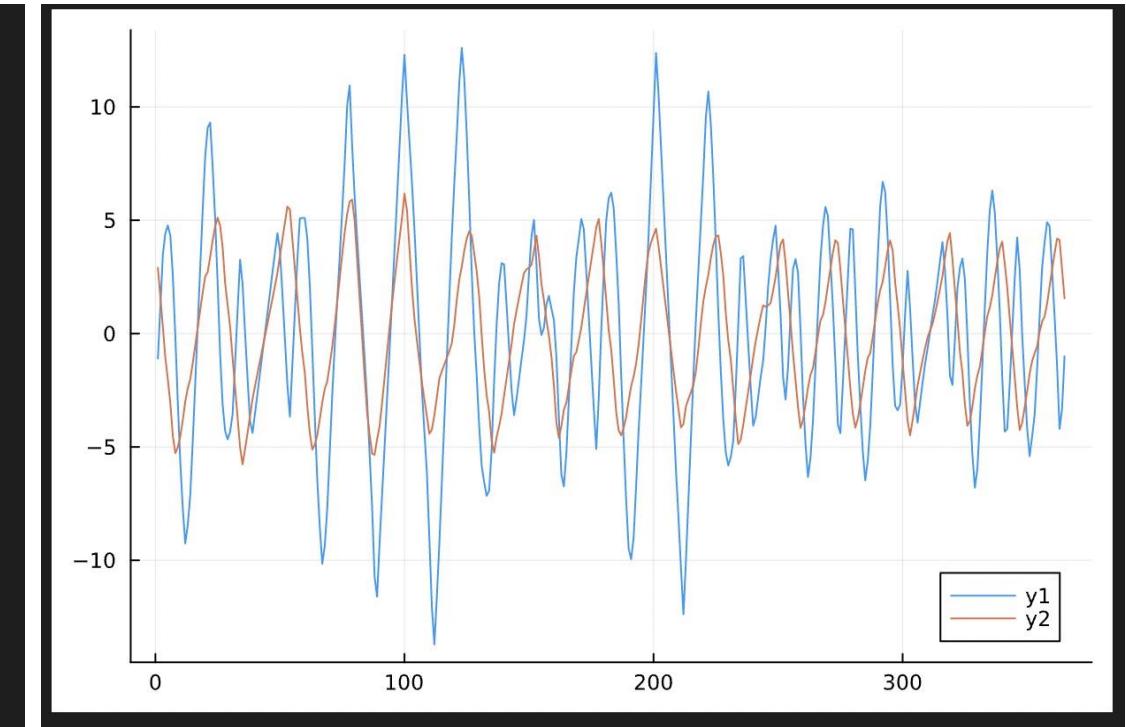
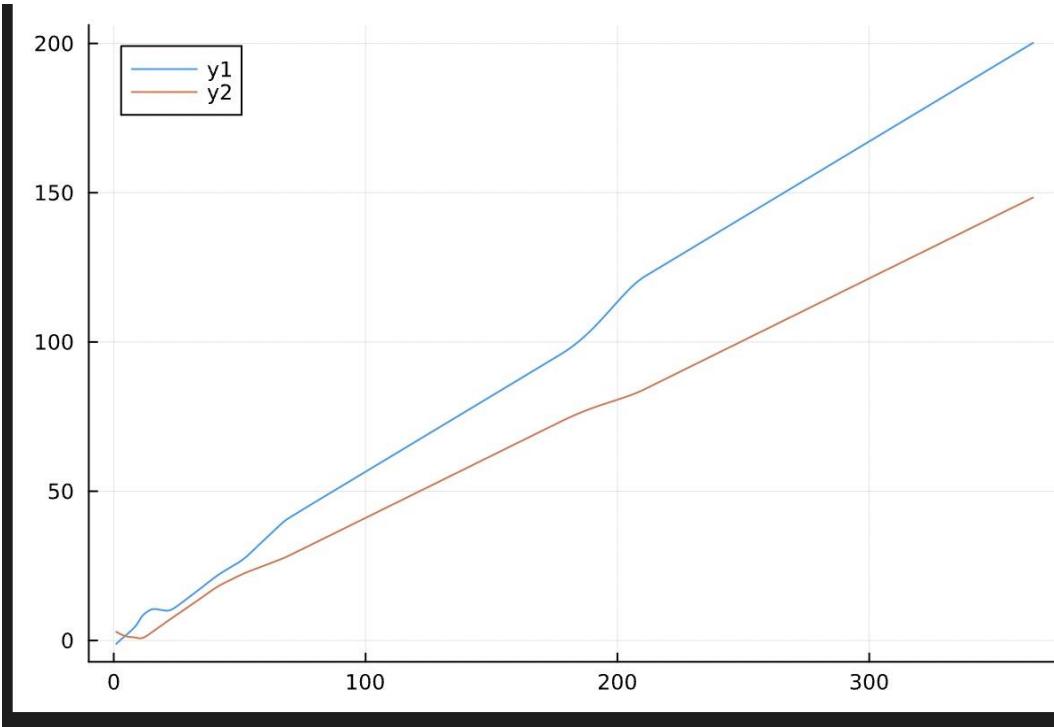
- コード

- u0=[-1.3,10.5]
- du0=[-1.1,2.9]
- tspan=(0,364)
- model=FastChain((x,p)-> x, FastDense(2,50,tanh),FastDense(50,10,tanh),FastDense(10,2))
- p = initial\_params(model)
- ff(du,u,p,t) = model(u,p)
- prob = SecondOrderODEProblem{false}(ff, du0, u0, tspan, p)
- sol=solve(prob,p=p,saveat=1)
- **function** loss\_n\_ode(p)
- pred = Array(solve(prob,p=p,saveat=1))
- sum(abs2, dataset .- pred[3:4, :]), pred
- **end**

- res = DiffEqFlux.sciml\_train(loss\_n\_ode, p, Adam(0.1), maxiters = 100)
- sol1=solve(prob,p=res,saveat=1)

# シミュレーション



# Diffeqライブラリの微分方程式法について まとめ

- 結論：この方法は予測できない　おすすめもしない
- 理由：まずこの方法の前提として微分方程式数学のモデルできるだけ細かくかつ正しくでなければならない　正直言って私はそのような数学知識はない
- それからこのDifferentialEquationsライブラリの資料は大体2、3年前　更新につれて使えない機能が多い公式サイトの例も古い（例えば最適化の機能とか）　使いにくい
- 気温 자체が微分方程式で表せるとしもノイズ入りあるいは確率微分方程式でやらなければならぬと思う（前に言った通りそのような数学知識はない）

## 方法2 fluxライブラリ中心の機械学習

- この方法ではjuliaのflux.jl(juliaの機械学習専用のライブラリ)を中心にニューラルネットワークを使って近似してデータを計算して予測として結果を出す
- 考えた方法は二つ  $x_{train}$  2021年1-364日のデータ  $y_{train}$  2-365 最後にテストは2022年のデータを使う
- あるいは直接 2021年と2022のデータを  $x, y$  にする 2021と2020のデータを学習させ 近似したニューラルネットワークを2021のデータを入力して2022の予測する
- Lstm gru transformer を使って近似して予測してみる
- Juliaの機械学習のためデータを全部Float32の  $m \times n$  の行列にする  $n$  は365あるいは364  $m$  は1あるいは2(日数と使うどの気温を使うかにより決定)

# データの処理について（以下のコードに従う）

- begin
- data=CSV.read("w2021-1.csv",DataFrame,header=6)
- data0=CSV.read("w2022-1.csv",DataFrame,header=6)
- data1=CSV.read("w2020.csv",DataFrame,header=6)
- dataset2021=Array(transpose(Matrix{Float32}(data[1:365,[5]])))
- dataset2022=Array(transpose(Matrix{Float32}(data0[1:365,[5]])))
- dataset2020=Array(transpose(Matrix{Float32}(data1[1:365,[4]])))
- x0=Array(transpose(dataset2020[1:364]))
- y0=Array(transpose(dataset2020[2:365]))
- x1=Array(transpose(dataset2021[1:364]))
- y1=Array(transpose(dataset2021[2:365]))
- x2=Array(transpose(dataset2022[1:364]))
- y2=Array(transpose(dataset2022[2:365]))
- end
- 結果は全部1かけるnのmatrix

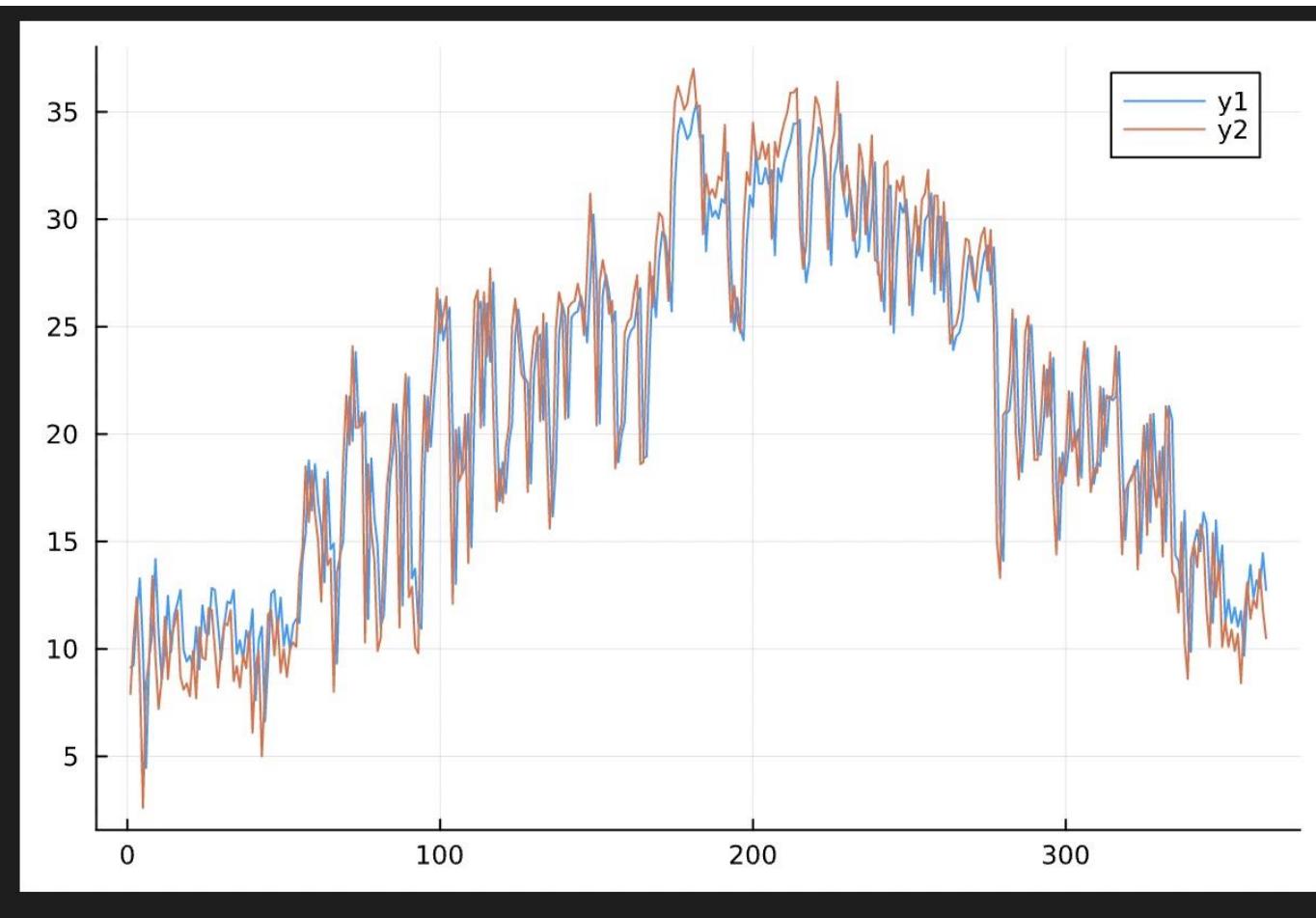
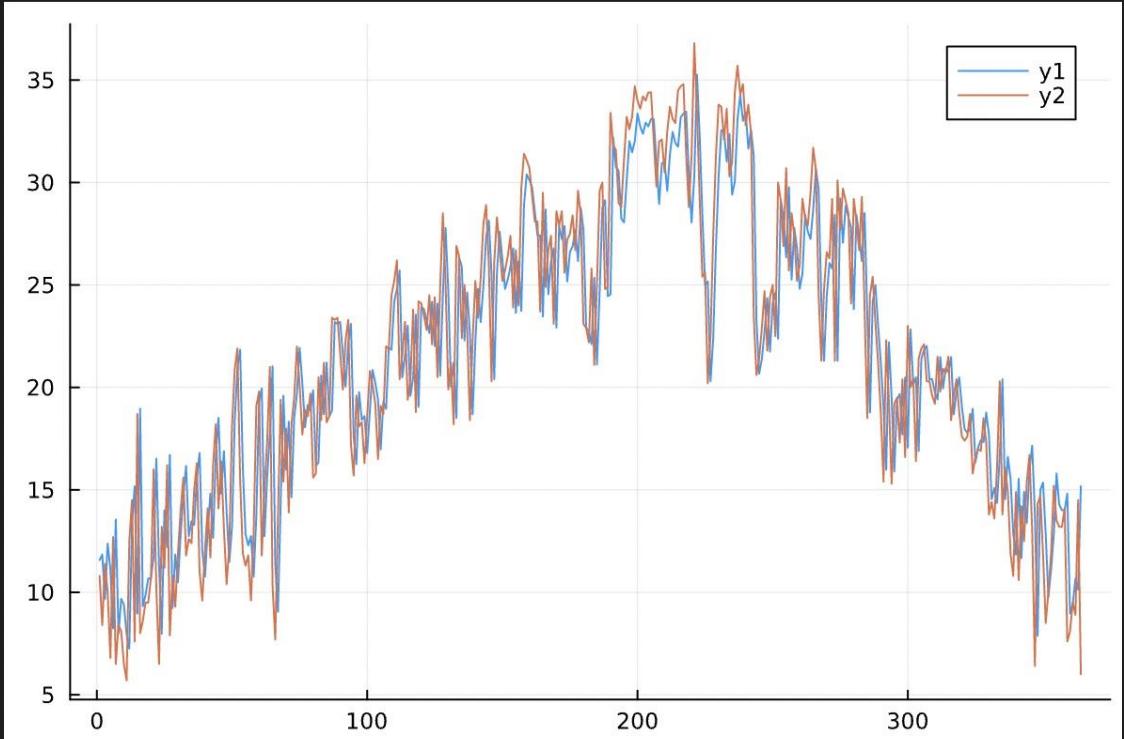
# まずはmlp

- 初めてのモデルは線形mlp
- model=Chain(Dense(1,20),Dense(20,30),Dense(30,1))
- x,yの訓練データは=data2021[1:364],data2021[2:365]  
data2020,data2021二つの方法で試してみる
- バッチを作らないそのまま計算していく
- 損失関数はmseを使う 最適化はadam
- 予測した結果を評価するために損失関数で数値の違い コサイン類似度でパターンが当てるかどうかを使う（コサインを使用した理由：他の類似度計算の方法がわからないかつコサインは投影としえ考えられて線形関係を表せるから）

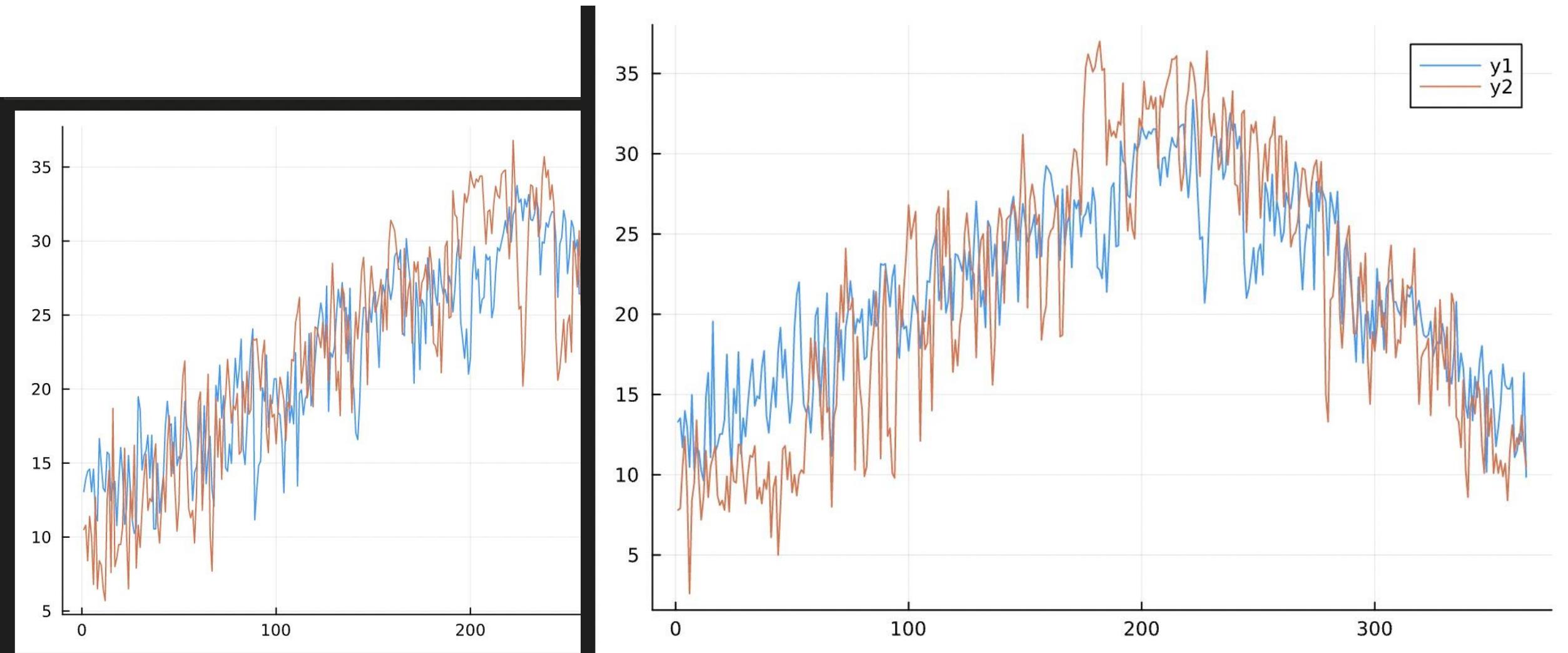
# 結果

- $x, y$  のトレーニングデータは [1:364] [2:365] の場合:
- トレーニングデータの類似度 : 0.9891083378544722
- 近似した結果の損失と類似度: 10.643116186549802,  
0.9895775557513834
- 予測の結果の損失と類似度 : (10.831873178986937,  
0.9894942633539136)
- $x, y$  のトレーニングデータは 2020 2021 の時:
- (20.232603937762235, 0.9767438467409213)
- 近似 : (20.232603937762235, 0.9800486243076176)
- 結果 : (24.578911503430202, 0.9759553099494683)

# [1:364] [2:365]の場合の近似と予測の可視化



# 2020 2021の場合



コード

- `using Flux,Optim,Plots,CSV,DataFrameLinearAlgebra`
- `model=Chain(Dense(1,20),Dense(20,30),Dense(30,1))`
- `opt=Adam(0.01)`
- `loss(x, y) = Flux.Losses.mse(model(x), y)`#損失関数
- `for i =1:2000`
- `grad = gradient(()>loss(x1,y1),Flux.params(model))`
- `Flux.update!(opt, Flux.params(model), grad)`
- `end`
- `cos_s(x,y)=sum(x.*y)/(norm(x)*norm(y))`#cos

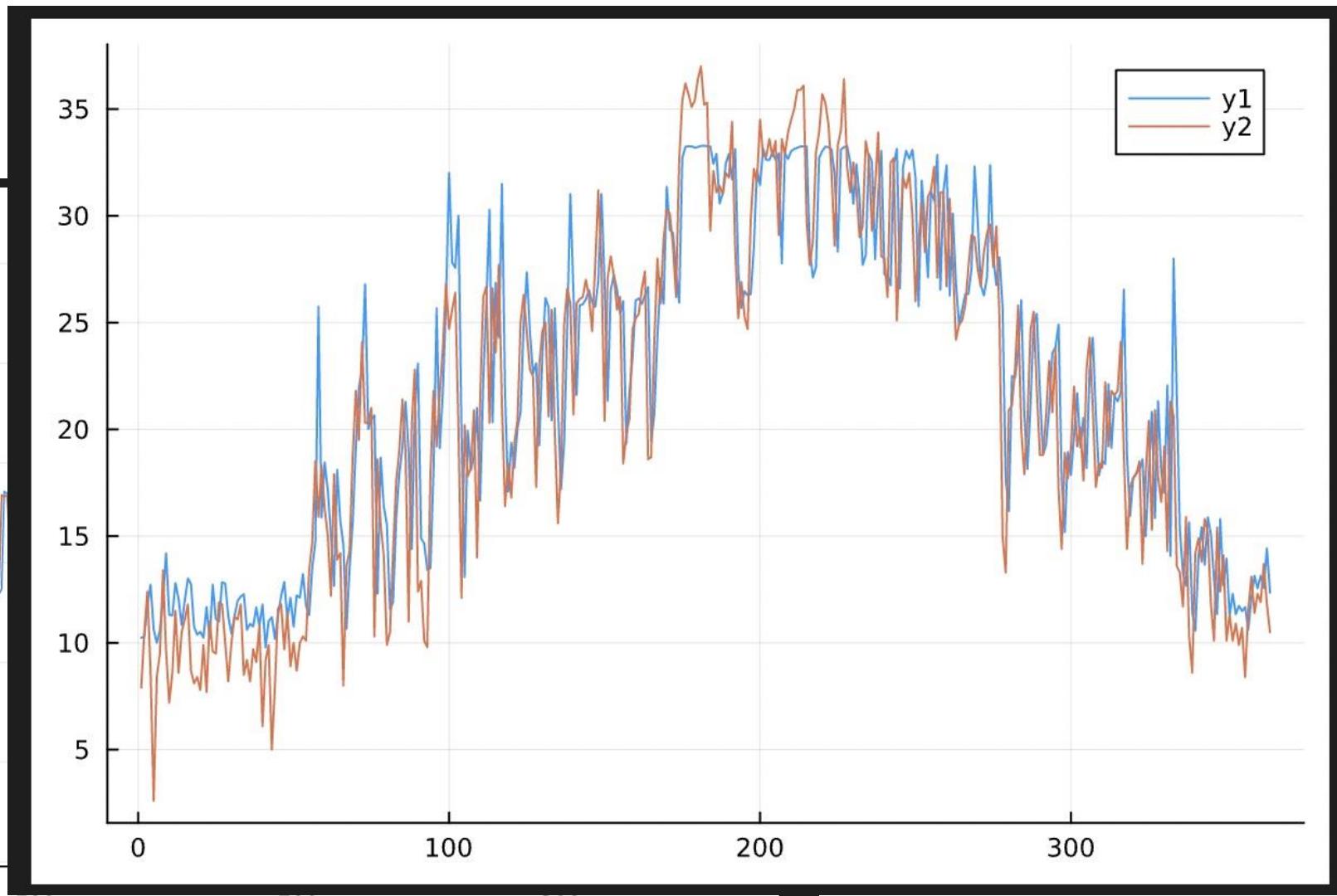
# 自然言語処理ニューラルネットワーク rnn

- 理由:このデータは一応時シリーズで自然言語処理用のニューラルネットワークと相性がいいはずだと思う
- まずはlstmを試す できるだけ簡単なlstm使う
  - model=Chain(LSTM(1 => 10), Dense(10 => 1))
  - 損失関数はmseのままにする。計算の手順は前と同じにする
  - それからgruも試す
  - model=Chain(GRU(1 => 10), Dense(10 => 1))
  - X,yは以前と同じ2種類の方法にする
  - それからぞれぞれで前の微分方程式法でやってみる

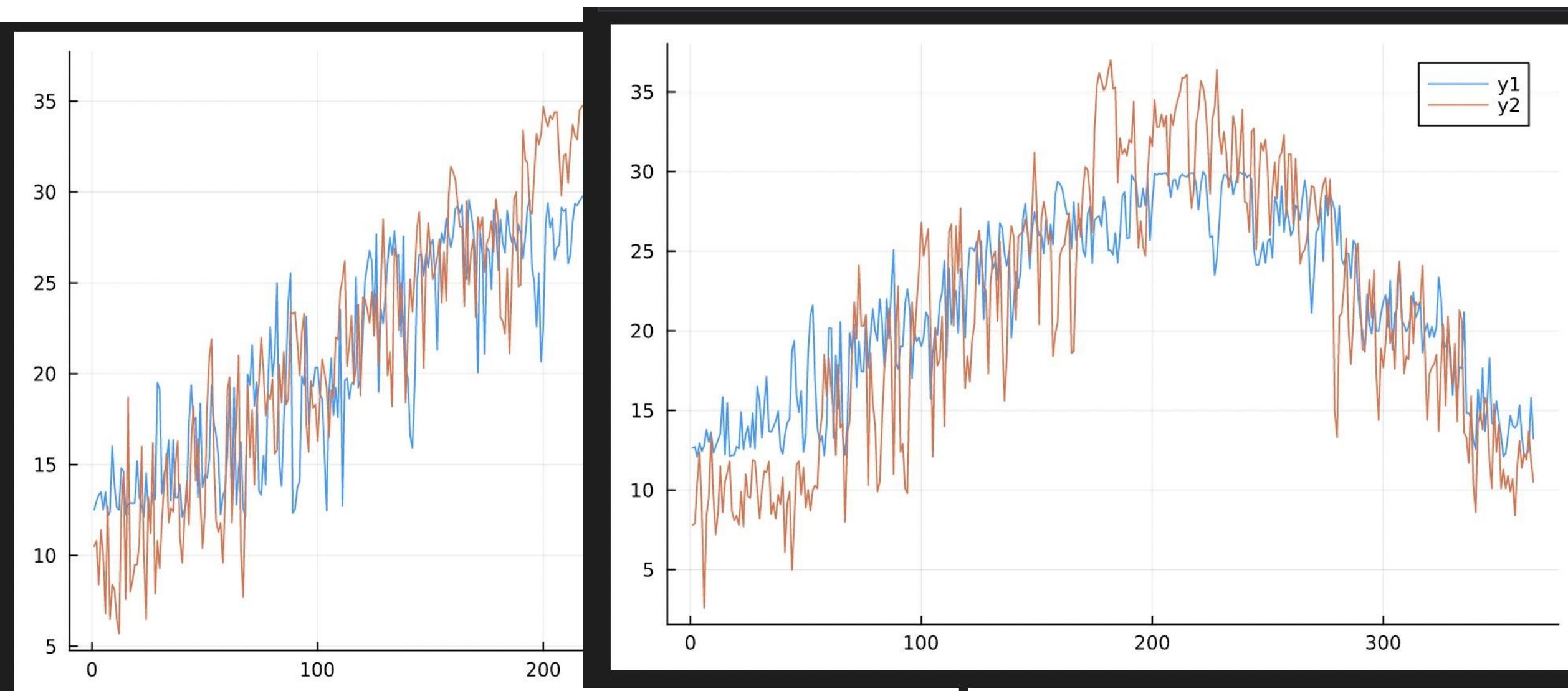
# Lstm

- [1:364] [2:365]の場合:
- 近似した結果の損失と類似度
  - (10.178420066833496, 0.9900352954864502)
- 予測の結果の損失と類似度
  - (11.294672012329102, 0.98915034532547)
- 2020 2021の時:
  - 近似した結果の損失と類似度
    - (19.07485008239746, 0.9812013506889343)
  - 予測の結果の損失と類似度
    - (22.383136749267578, 0.9779164791107178)

# [1:364] [2:365]の場合の近似と予測の可視化



# 2020 2021 の 時



# コード

using

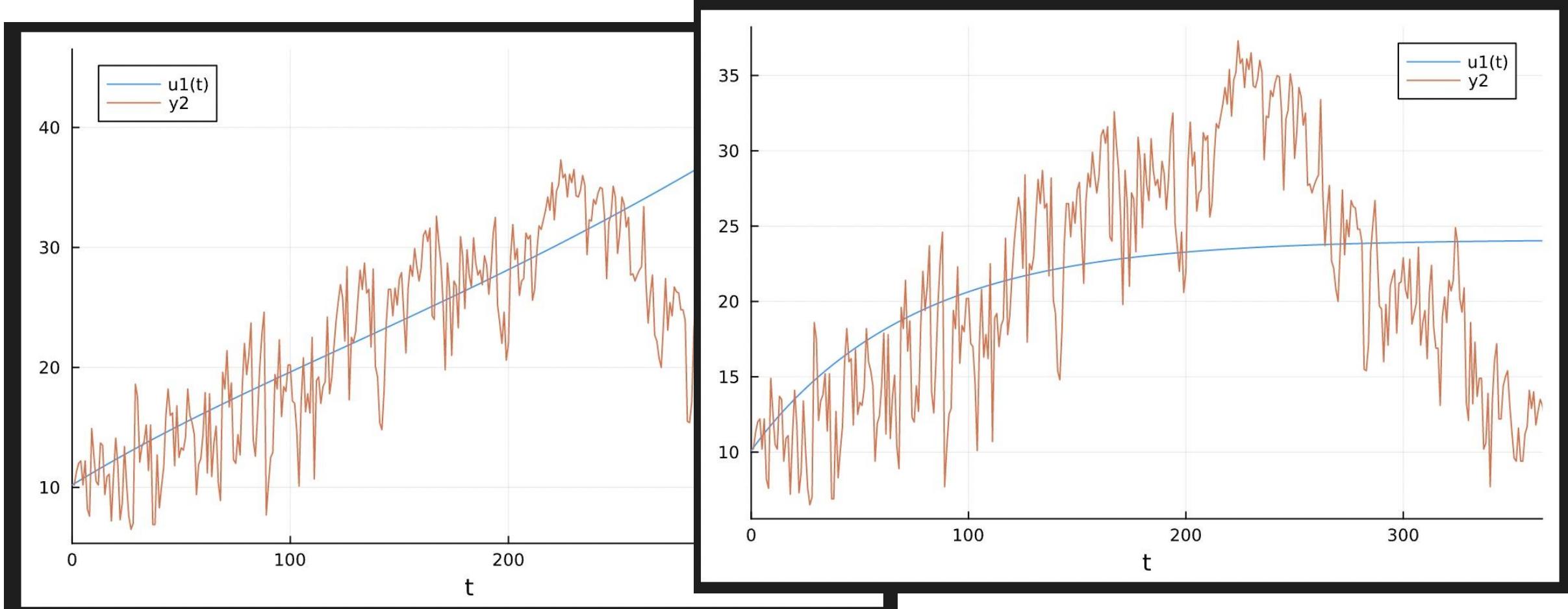
DiffEqFlux, OrdinaryDiffEq, Flux, Optim, Plots, DifferentialEquations, CSV, DataFrames,

- using ForwardDiff, LinearAlgebra
- model=Chain(LSTM(1 => 10), Dense(10 => 1))
- for i =1:2000
- grad = gradient(()->loss(dataset2020, dataset2021),  
Flux.params(model))
- Flux.update!(opt, Flux.params(model), grad)
- end

# Lstmと微分方程式連携 $\frac{dT}{dt} = \text{Lstm}(T)$

- 前の微分方程式の方法と同じ 損失関数はmse コードは以下
- dudt2=Chain(LSTM(1 => 10), Dense(10 => 1))
- tspan = (Float32(0.0),Float32(364))
- p,re = Flux.destructure(dudt2)
- dudt(u,p,t) = re(p)(u)
- u0=[dataset2020[1]]
- prob = ODEProblem(dudt,u0,tspan)
- **for** i =1:2000
- grad = gradient(()->loss(dataset2020,dataset2021), Flux.params(u0,p))
- Flux.update!(opt, Flux.params(u0,p), grad)
- **end**
- sol1=solve(prob,Tsit5(),u0=u0,p=Flux.params(u0,p)[2],saveat=1)

Lstmと微分方程式連携  $\frac{dT}{dt} = \text{Lstm}(T)$  の結果可視化  
2020のデータを使って



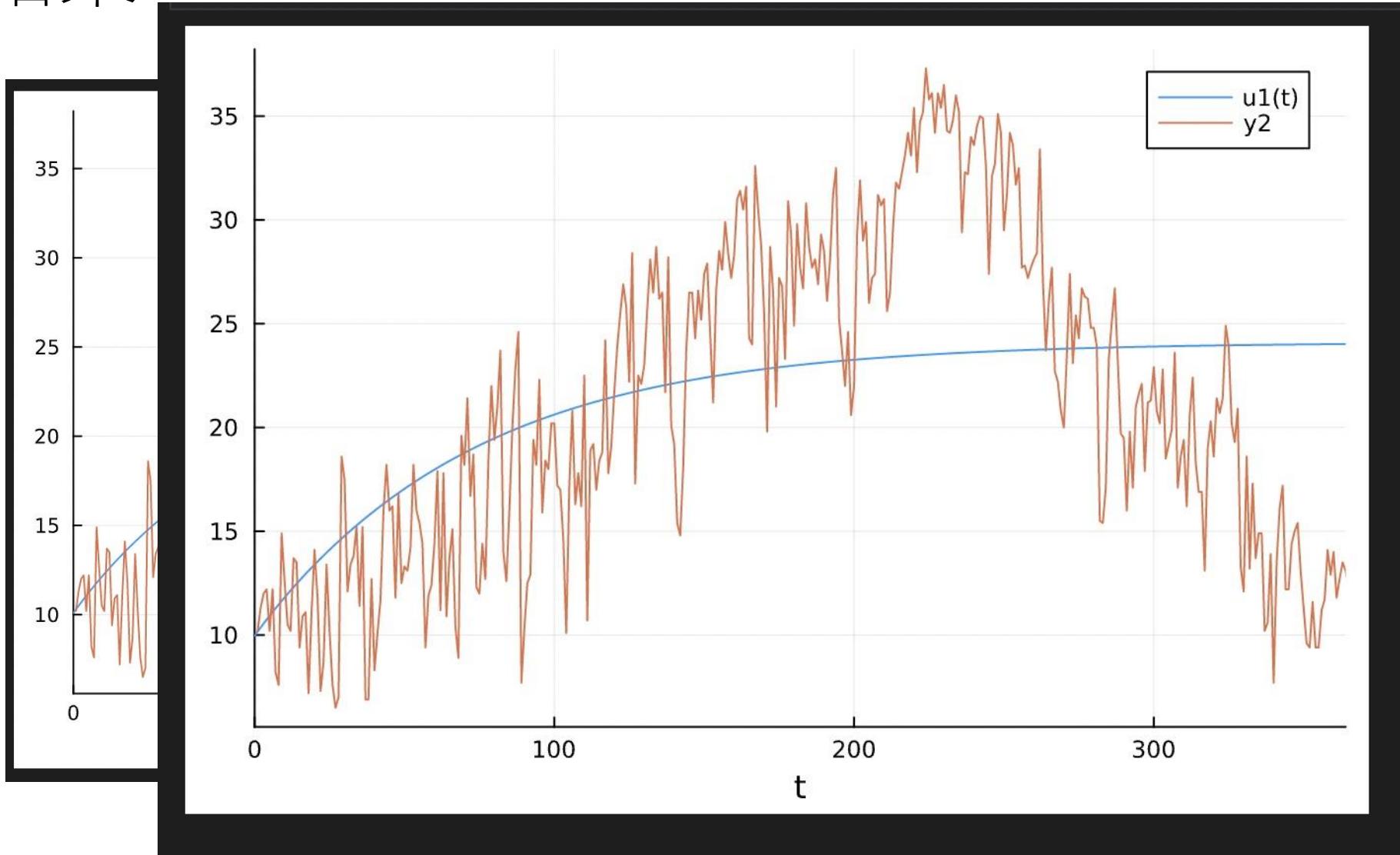
gru

- [1:364] [2:365]の場合:
  - 近似した結果の損失と類似度
    - (11.950977325439453, 0.9901505708694458)
  - 予測の結果の損失と類似度
    - (10.453295707702637, 0.990019679069519)
- 2020 2021の時:
  - 近似した結果の損失と類似度
    - (19.67729377746582, 0.9806019067764282)
  - 予測の結果の損失と類似度
    - (24.957820892333984, 0.9755512475967407)

# gruと微分方程式連携 $\frac{dT}{dt} = GRU(\tau)$

- 前と同じ
- begin
- dudt2=Chain(GRU(1 => 10), Dense(10 => 1))
- tspan = (Float32(0.0),Float32(364))
- p,re = Flux.destructure(dudt2) # use this p as the initial condition!
- dudt(u,p,t) = re(p)(u) # need to restrcture for backprop!
- u0=[dataset2020[1]]
- prob = ODEProblem(dudt,u0,tspan)
- function loss(x)
- pred=Array(solve(prob,Tsit5(),u0=u0,p=p,saveat=1))
- l=sum(abs2,(pred[2:365].-x[2:365]))/length(364)
- end
- for i =1:1000
- grad = gradient(()->loss(dataset2020), Flux.params(u0,p))
- Flux.update!(opt, Flux.params(u0,p), grad)
- end
- sol1=solve(prob,Tsit5(),u0=u0,p=Flux.params(u0,p)[2],saveat=1)
- plot(sol1)
- plot!(transpose(dataset2020))
- end

# 結果



# Transformers

使うモデルは右の定番のtransformersのモデル 今回はencoderとdecoderはそれぞれ二つにする

今回はデータを分けないそのまま入力する パラーメーターは全部定番のを使う

JuliaのTransfomersとflux二つのライブラリーを連動する

構造は以下になる

```
encoder_input_layer = Dense(1,512)
decoder_input_layer = Dense(1,512)
encode_t1 = Transformer(512, 8, 64, 2048)
encode_t2 = Transformer(512, 8, 64, 2048)
decode_t1 = TransformerDecoder(512, 8, 64, 2048)
decode_t2 = TransformerDecoder(512, 8, 64, 2048)
positional_encoding_layer = PositionEmbedding(512)
linear = Dense(512,1)
p = 0.2
dropout_pos_enc = Dropout(p)
```

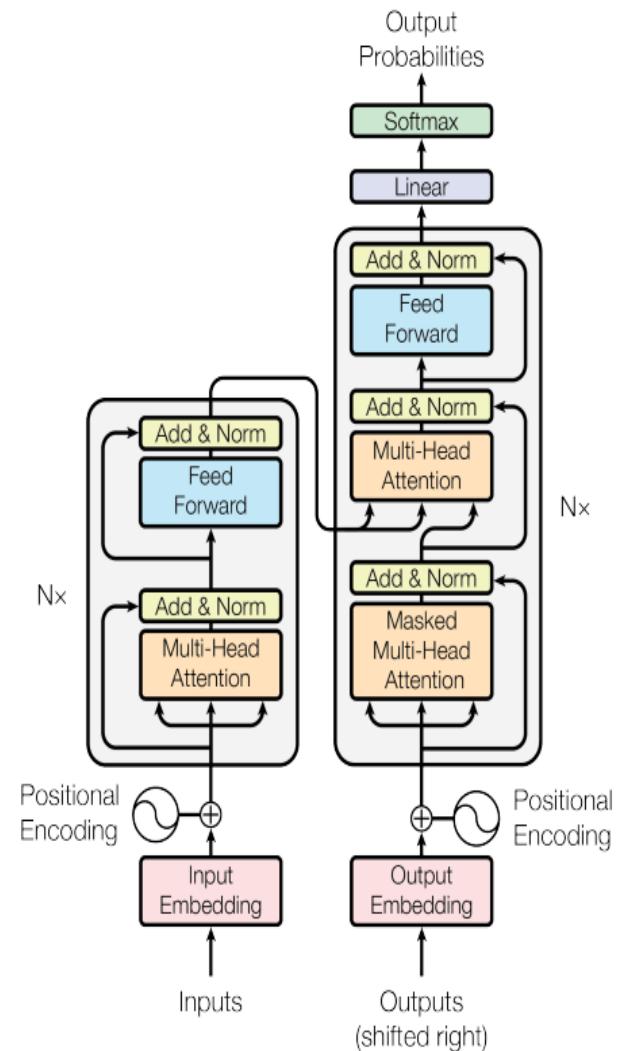


Figure 1: The Transformer - model architecture.

# Forward と loss のコード

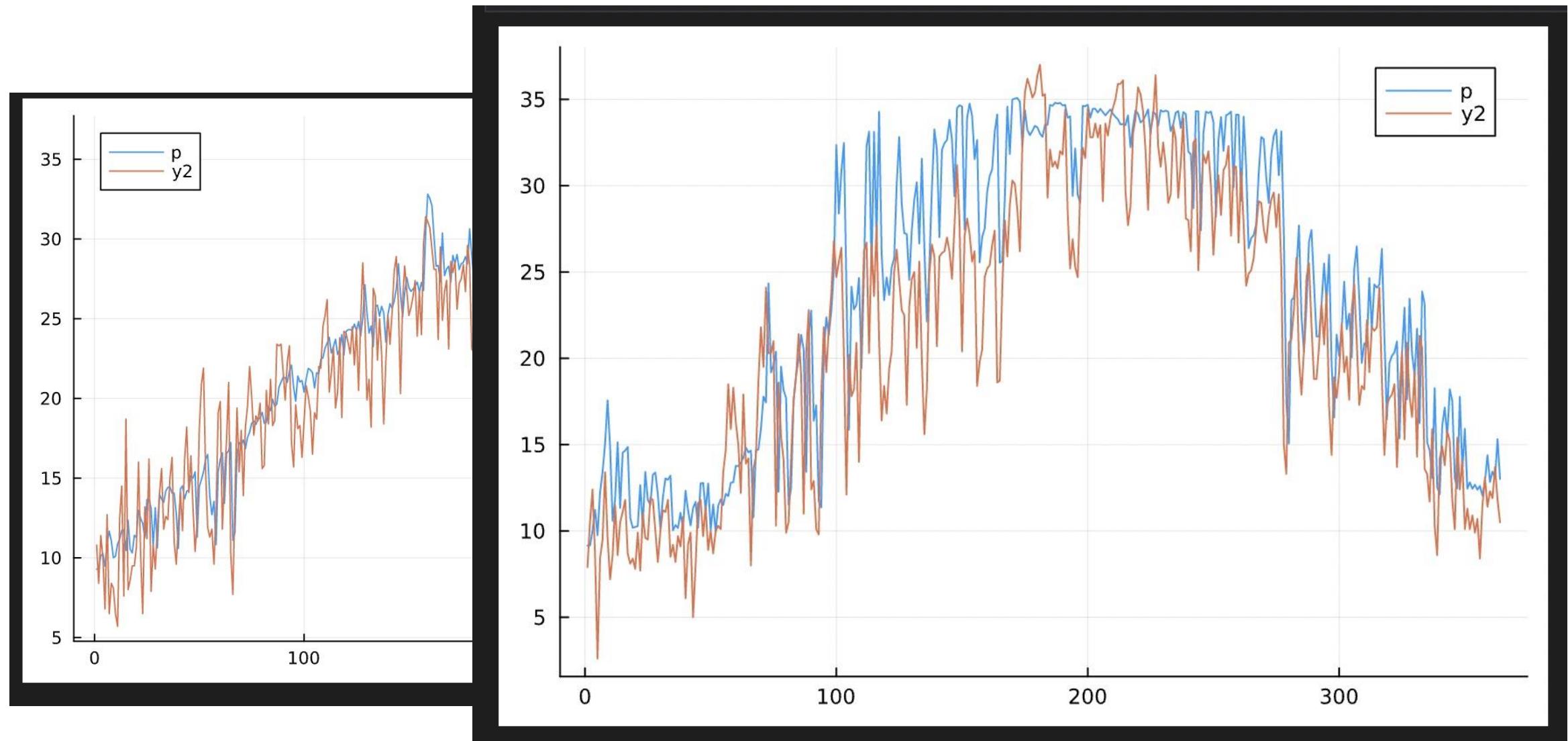
- `function encoder_forward(x)`
- `x = encoder_input_layer(x)`
- `e = positional_encoding_layer(x)`
- `t1 = x .+ e`
- `t1 = dropout_pos_enc(t1)`
- `t1 = encode_t1(t1)`
- `t1 = encode_t2(t1)`
- `return t1`
- `end`
- `function decoder_forward(tgt,t1)`
- `decoder_output =decoder_input_layer(tgt)`
- `t2 = decode_t1(decoder_output,t1)`
- `t2 = decode_t2(decoder_output,t2)`
- `t2=Flux.flatten(t2)`
- `p = linear(t2)`
- `return p`
- `end`

```
function loss(src, trg, trg_y)
enc = encoder_forward(src)
dec = decoder_forward(trg, enc)
err = Flux.Losses.mse(dec, trg_y)
return err
end
```

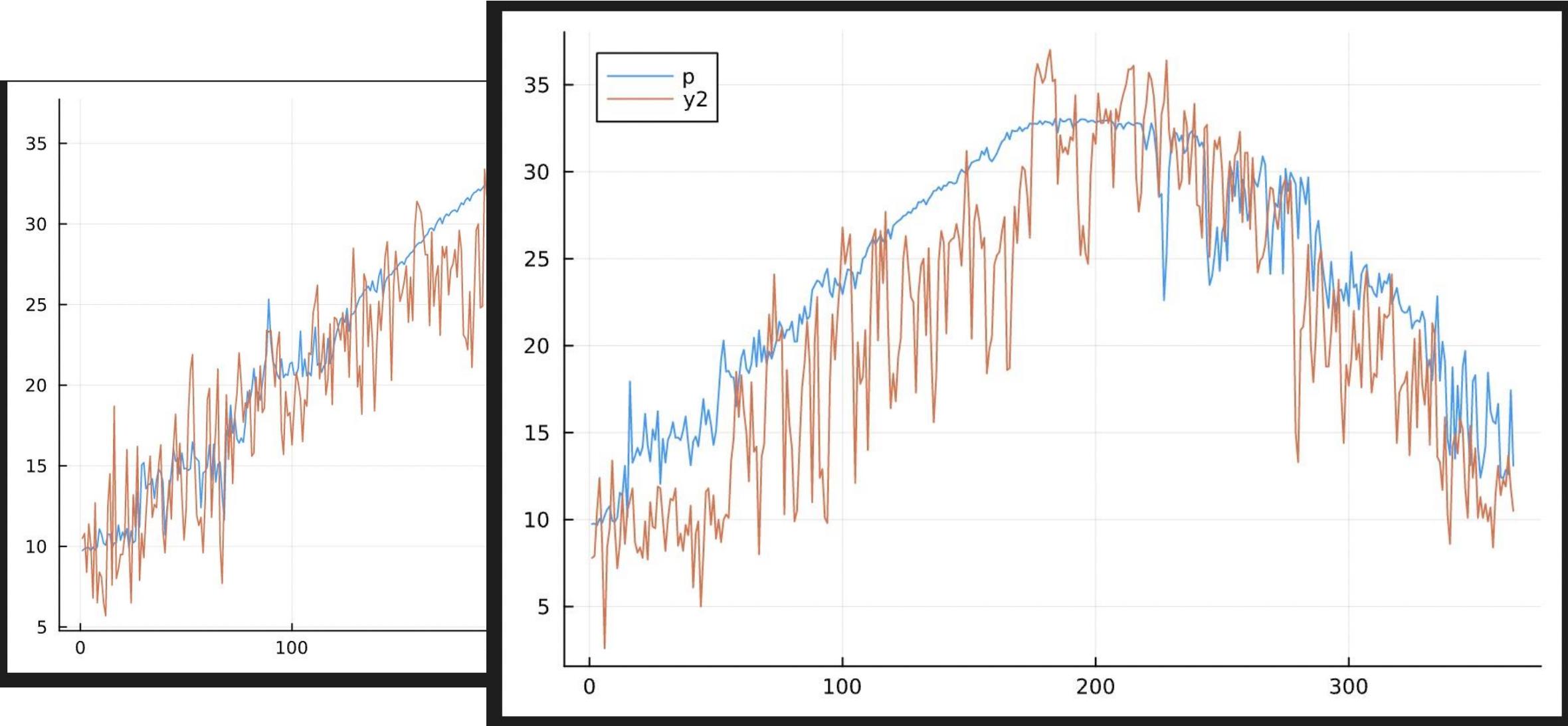
# Transformers

- [1:364] [2:365]の場合:
- 近似した結果の損失と類似度
  - (9.941707611083984, 0.9919296503067017)
- 予測の結果の損失と類似度
  - (21.97441864013672, 0.9866981506347656)
- 2020 2021の時:
  - 近似した結果の損失と類似度
    - (13.968463897705078, 0.9885773062705994)
  - 予測の結果の損失と類似度
    - (27.21100616455078, 0.9812406301498413)

# [1:364] [2:365]の場合の近似と予測の可視化



# 2020 2021 の 時



# Rnnについてのまとめ

- 結論としてはgruが一番いい 損失小さく類似度も高い
- 今回のx、y自体の類似度が高いためコサイン類似度を計算したところで0.0nnnnnnnnの差しかない データ実際の数値は約1-30あまり差がない
- 損失関数だけ測るがいい
- 微分方程式の方法から見ると半年の気温が段々上がる傾向はシミュレーションできた rnnはこの気温のデータのモデルとして相性がいいと思う
- Transfomersの近似効果は一番いい
- Lstm is dead long live the transfomers !

# 方法 3 Turing, AbstractGPs,を中心の確率モデル法

- ベイズニューラルネットワークとガウス過程とdeep kernel learningを使う
- 問題を簡単するため 全部ベイズのガウス推論を使う (理由はガウスかけるガウスはガウス 投影もガウス 周辺確率の積分もガウス こうすると結果もガウスで表せる)

# ベイズニューラルネットワーク

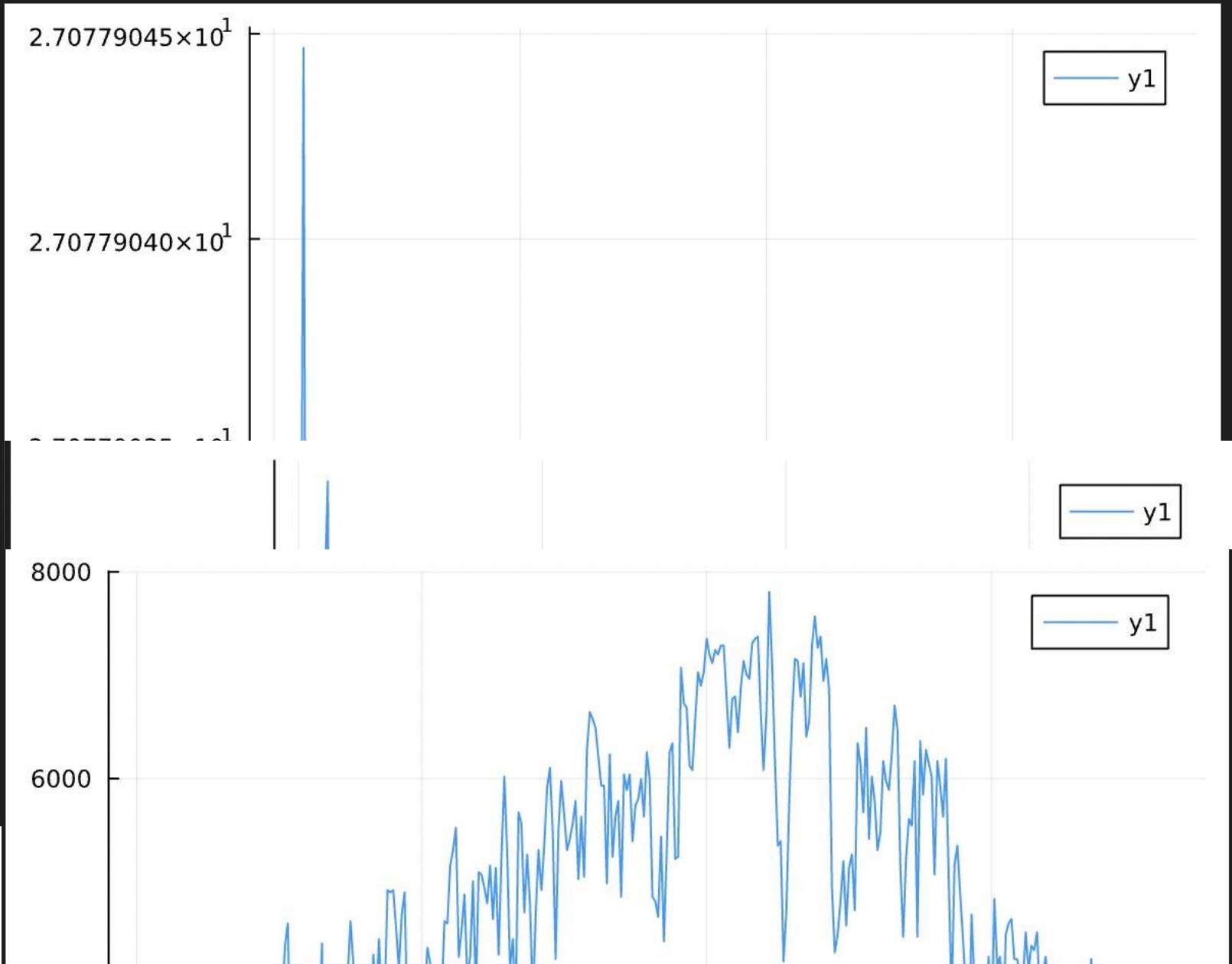
- この方法はニューラルネットワークのパラメータをベイズで更新する方法
- 事前確率prior=nnのパラメータの確率( $p(\theta)$ ) 
$$p(\theta|D) = \frac{p(D|\theta)p(\theta)}{p(D)}$$
- 尤度likelihoodは nnの出力と関連するある分布の確率 $p(D|\theta)$
- Julia のturing ライブラリではこの二つを設置していれば他の数値が自動的に計算してくれる
- 一般に事前確率prior~N(0,1)にする
- 尤度likelihood ~Normal(NN(xi),Sigma)にする
- この方法でパラメータを更新する
- Juliaのコードはほぼ数学式と同じなのでコードでモデルを見せる

# 流れ

- まずはニューラルネットワークを定義
- 確率モデルも定義する
- `@model function bayes_nn(xs, ts, nparameters, reconstruct; alpha=0.09)`
- `parameters ~ MvNormal(Zeros(nparameters), I / alpha)`
- `nn = reconstruct(parameters)`
- `preds = nn(xs)`
- `for i in 1:length(ts)`
- `ts[i] ~ Normal(preds[i],1)`
- `end`
- `end;`
- パラメータは平均0,分散は単位行列をalphaで割るもの多次元正規分布に従う
- このパラメータでニューラルネットワークを再構築
- 後で尤度を $N(nn(x),1)$ にする(実際分散を他にしたところで成功はしなかった,これもできなかった)
- これらを設置してあとはturingライブラリに任せてサンプリングする `sampler`はhmcにする(微分方程式つきのモンテカルロ)
- 最後は更新したパラメータの数はサンプリングの回数個にあるから周辺分布の積分として平均で計算して結果を出す

# 結果の可視化

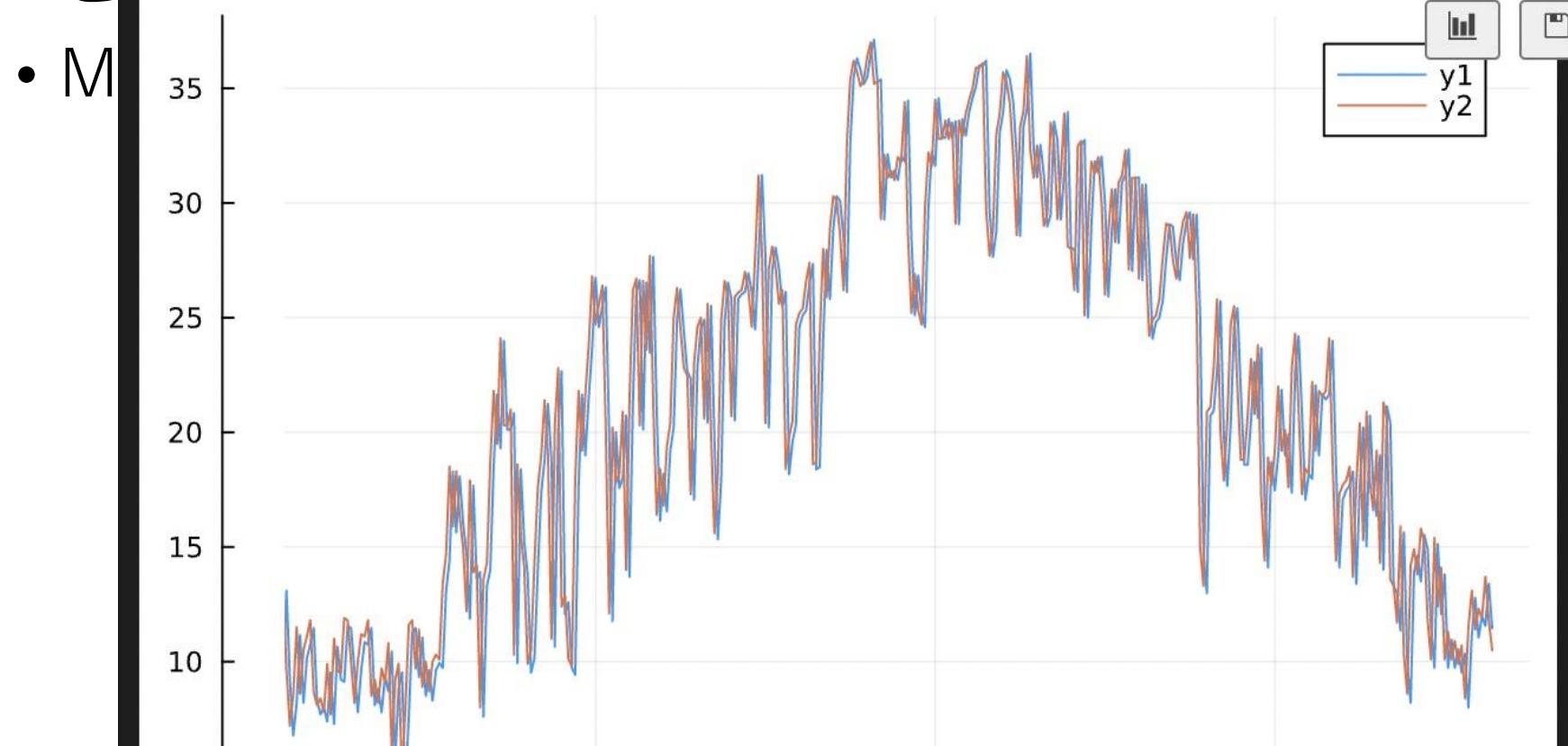
- Chain(Dense)
- Chain(Dense)
- Chain(Dense)
- 以上三つの



# 続き

- 前のグラフから見る線形mlpの時は一応近似した形状はほぼ同じ実際の数値は大きいが それなら全体を出力した結果のノルムで割りまた入力データのノルムを掛けばいいかもしれない

- この辺で仕事もマジでアツ



- `@model function bayes_nn(xs, ts, nparameters,reconstruct; alpha=0.09)`
- `parameters ~ MvNormal(Zeros(nparameters), I /0.09)`
- `nn = reconstruct(parameters)`
- `preds = nn(xs)`
- `for i in 1:length(ts)`
- `ts[i] ~ Normal(preds[i],1)`
- `end`
- `end;`

```
nn_initial = Chain(Dense(1, 3, ), Dense(3, 2, ), Dense(2, 1,))  
parameters_initial, reconstruct = Flux.destructure(nn_initial)  
length(parameters_initial)
```

```
N = 1000  
ch = sample(  
    bayes_nn(x1, y1, length(parameters_initial), reconstruct), HMC(0.05, 1), N  
);  
theta = MCMCChains.group(ch, :parameters).value;  
nn_forward(x, theta) = reconstruct(theta)(x)  
r1=[nn_forward(x2,theta[i,:]) for i in 1:1:1000]  
re1=mean(r1)  
re2=(re1./norm(re1).* (norm(x1)))
```

$$k(x, x') = \exp\left(-\frac{d(x, x')^2}{2}\right).$$

# ガウス過程

By default,  $d$  is the Euclidean metric  $d(x, x') = \|x - x'\|_2$ .

$$f(x) := \phi_x^\top w \quad p(w) = \mathcal{N}(w; \mu, \Sigma) \quad p(y | f) = \mathcal{N}(y; f_x, \sigma^2 I)$$

$$p(w | y, \phi_x) = \mathcal{N}\left(w; \mu + \Sigma \phi_x (\phi_x^\top \Sigma \phi_x + \sigma^2 I)^{-1} (y - \phi_x^\top \mu), \Sigma - \Sigma \phi_x (\phi_x^\top \Sigma \phi_x + \sigma^2 I)^{-1} \phi_x^\top \Sigma\right)$$

$$p(f_x | y, \phi_x) = \mathcal{N}\left(f_x; \phi_x^\top \mu + \phi_x^\top \Sigma \phi_x (\phi_x^\top \Sigma \phi_x + \sigma^2 I)^{-1} (y - \phi_x^\top \mu), \phi_x^\top \left(\Sigma - \Sigma \phi_x (\phi_x^\top \Sigma \phi_x + \sigma^2 I)^{-1} \phi_x^\top \Sigma\right) \phi_x\right)$$

：れているものに従  
、

- カーネルは色々なものがあるが今回はSqExponentialKernelだけ使う
- あとは公式に記載されたと同じARDTransformを入れる
- ARDTransform( $\alpha$ )の仕組みは中にjuliaのデータタイプのvectorをにれることで入力するデータを先入力したvectorにかける
- こうなると事前のパラメータは $\alpha$ と尤度の $\Sigma$
- 事前確率  $\alpha \sim \text{MvLogNormal}(\text{MvNormal}(\text{zeros}, I))$
- $\sigma \sim \text{LogNormal}(0.0, 1.0)$
- ガウス過程 尤度を $y \sim \text{MvNormal}(\text{mean}(k), \text{cov}(k) + \Sigma \times I)$
- Kはカーネル  $k(x) = e^{-d(x, x')^2 / 2}$   $I$ は単位行列 (対角線は 1 他は 0 のmatrix)
- コレスキー分解できるように cov に  $1e-6 * I$  を足し算

$$\begin{aligned} & \phi_x^\top \Sigma \phi_x - \phi_x^\top \Sigma \phi_X (\phi_X^\top \Sigma \phi_X + \sigma^2 I)^{-1} \phi_X^\top \Sigma \phi_X \\ &= \mathcal{N}(f_x; \textcolor{teal}{m}_x + k_{xx}(k_{XX} + \sigma^2 I)^{-1}(y - \textcolor{teal}{m}_X), \\ & \quad k_{xx} - k_{xx}(k_{XX} + \sigma^2 I)^{-1}k_{xx}) \end{aligned}$$

り時間

using the abstraction / encapsulation

$$m_x := \phi_x^\top \mu$$

$$m : \mathbb{X} \rightarrow \mathbb{R}$$

mean function

出す

$$k_{ab} := \phi_a^\top \Sigma \phi_b$$

$$k : \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{R}$$

covariance function, aka. **kernel**

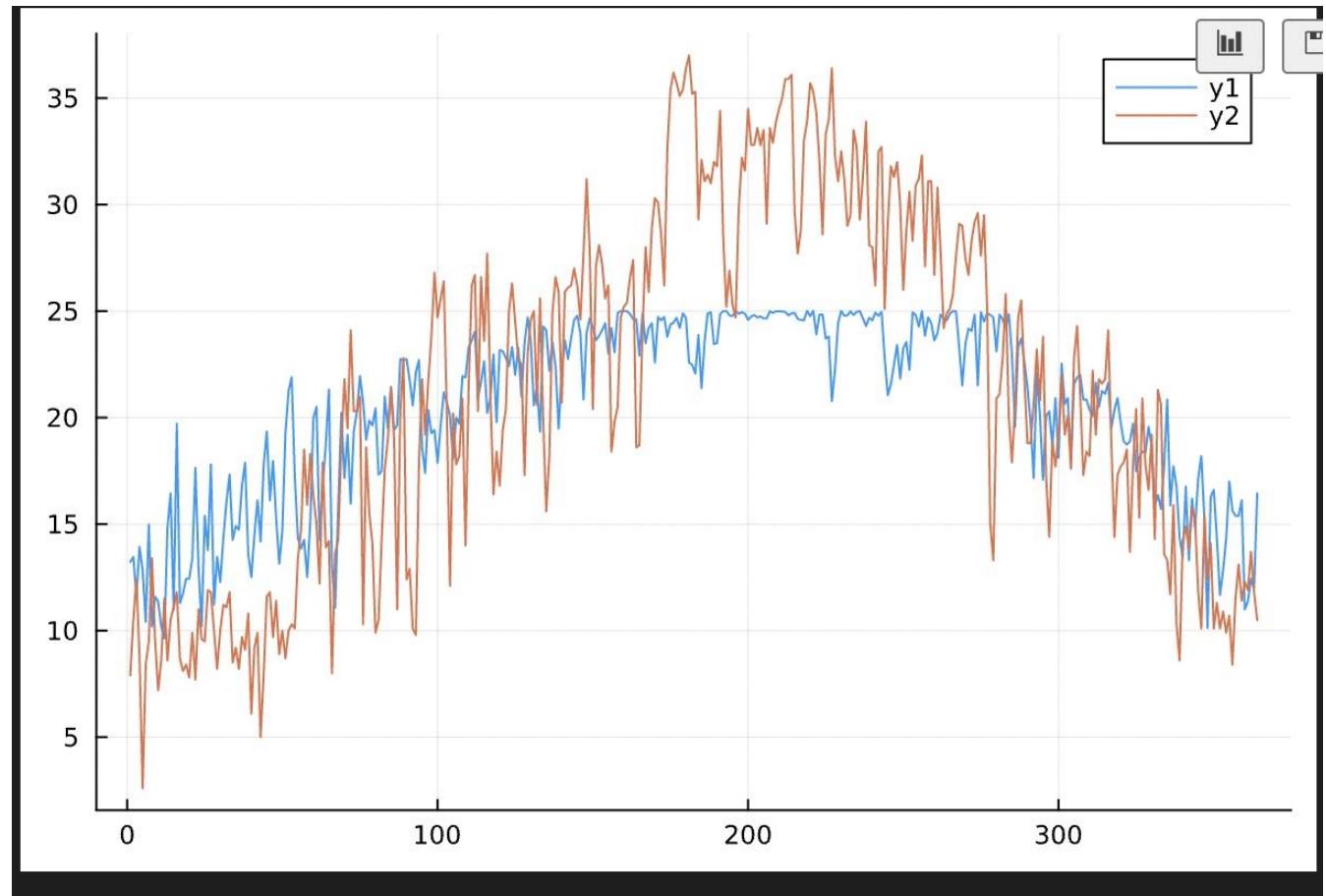
うと更  
今回は  
それぞ

ナレロリハス、ノハトノヘハツム

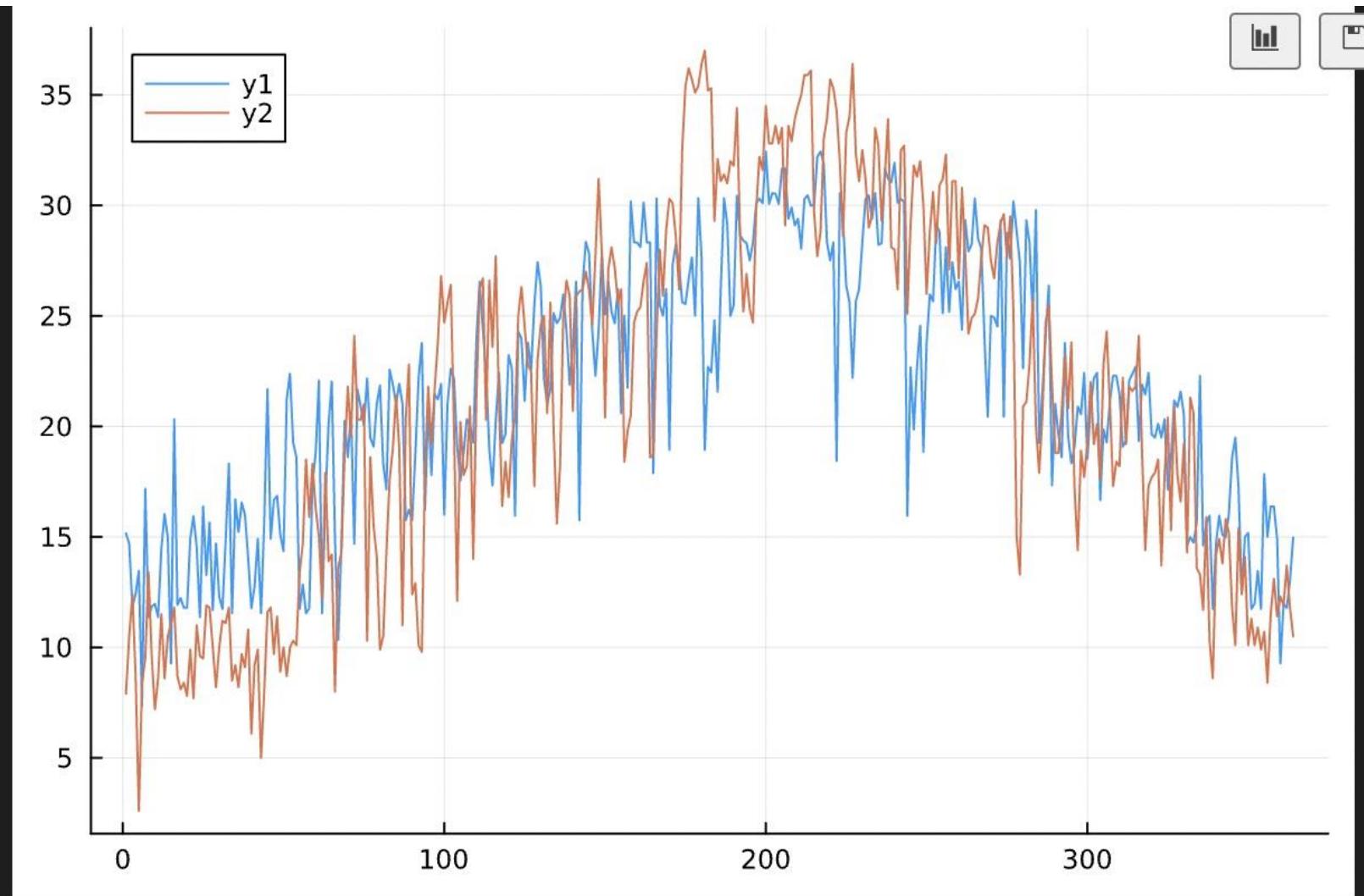
- しかしそのまま計算するとコレスキーフィルタができなくなることがある それで計算を大体その流れに沿い結果が出る方法を探した
- もし上記の公式に沿い最後はサンプル数のものの平均を計算したら次の図になる
- 計算の流れは訓練データのカーネルの平均を上記の  $m_x$   $k_{xx}$  は訓練データの共分散  $k_{XX}$  はテストデータの共分散
- $k_{xX}$  は二つデータの共分散 そのまま式で計算する サンプリング回数の結果を出す

# 結果

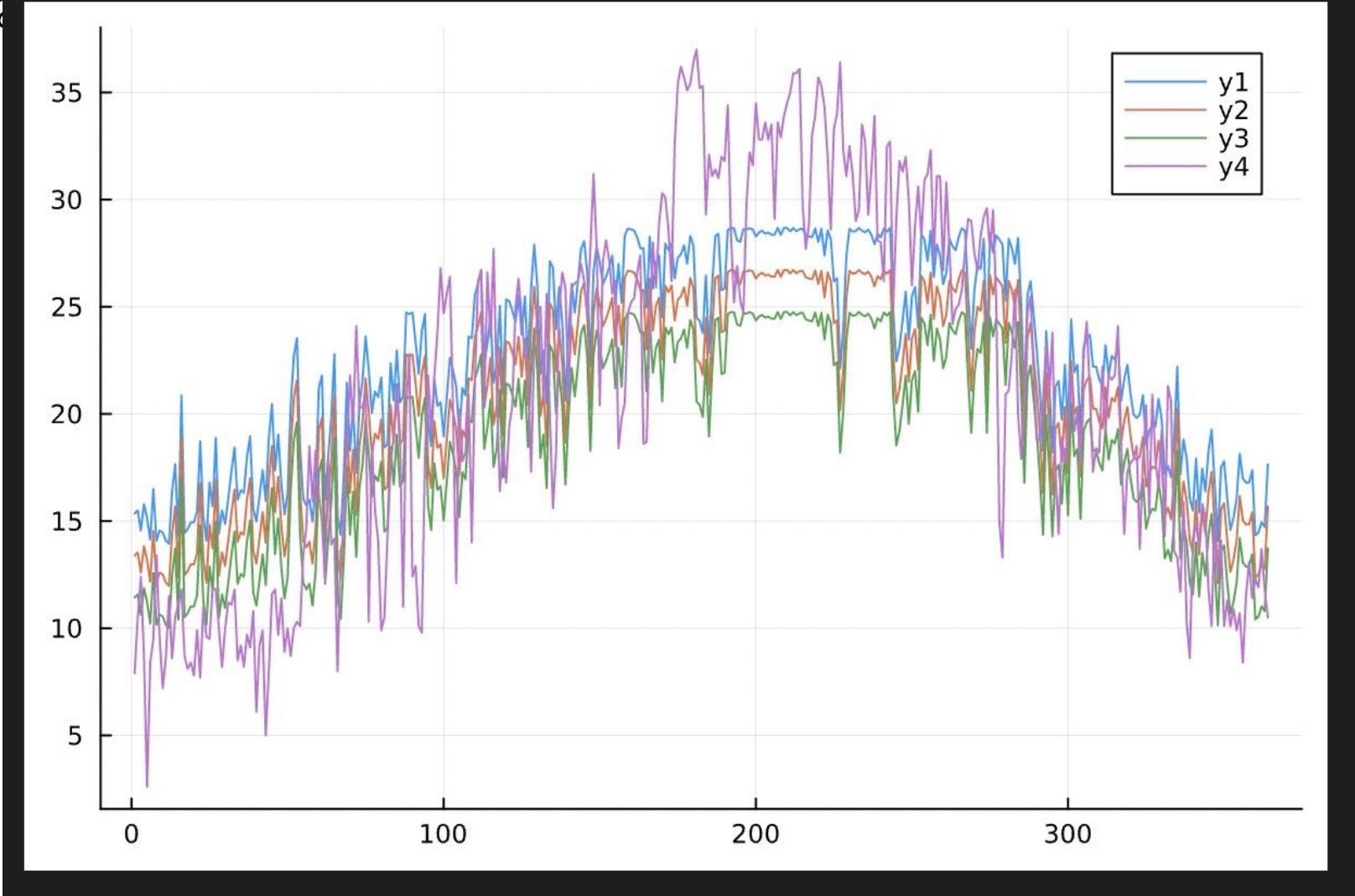
実際に平均を計算した結果は分散は小さい 結果のグラフは平均の平均の結果分散が小さいため多分区間を計算しても特に変わるものはないから計算しなかった



- 結果はそれなりに悪くないかもしだれが他の方法で結果を出す
- ネット



- 平均値を最初からランダムで決め最後は平均で[0.025,0.975]の区間を表す



— ⊥

- `sekernel(α) = SqExponentialKernel() ∘ ARDTransform(α);`
- `@model function GPLVM(x,y, K)`
- `noise = 1e-3`
- `α ~ MvLogNormal(MvNormal(Zeros(K), I))`
- `σ ~ LogNormal(0.0, 1.0)`
- `kxx=GP(sekernel(α))(ColVecs(x1),noise)`
- `y ~ MvNormal(mean(kxx), (cov(kxx)+1e-6 * I)+σ*I)`
- `return nothing`
- `end;`
- `gplvm = GPLVM(x1,vec(y1),1)`
- `chain_gplvm = sample(gplvm, HMC(0.01, 100), 500);`
- `mu = group(chain_gplvm, :α ).value.data[:, :, 1];`
- `sig2 = group(chain_gplvm, :σ).value.data[:, :, 1];`
- `Z=[x1 x2]`
- `kxx(α,x)=GP(sekernel(α))(x);`

- `function` `new`(`x,xnew,y,kxx,mu,sig`)
- `pre=kxx(mu,x)`
- `pre1=kxx(mu,[x xnew])`
- `kk=cov(pre1)`
- `kno=kk[1:364,365:end]`
- `c=inv(kk[365:end,365:end]+sig*I)`
- `m=mean(pre).+kno*c*(y-mean(pre))`
- `s=kno.-kno*c*kno'`
- `return (m,s)`
- `end`
- `nkk=[new(x1,x2,y1,kxx,[mu[i]],sig2[i]) for i in 1:length(mu)]`
- `mm=[nkk[i][1] for i in 1:length(nkk)]`
- `ss=[nkk[i][2] for i in 1:length(nkk)]`
- `mean(mm)`
- `ktx=GP(sekernel(vec(mu)))(ColVecs(Z),1e-3)`
- `kkk=cov(ktx);`
- `koinv=inv(kkk[365:end,365:end]);`
- `knn=kkk[1:364,1:364];`
- `kno=kkk[1:364,365:end];`
- `c=kno*koinv;`
- `m=c*(y1.-mean(kkk)).+mean(kkk);`
- `s=Matrix(LinearAlgebra.Hermitian(knn-c*kno));`
- `mvn=MvNormal(vec(m),s+(1e-6)*I+mean(sig2)*I(364))`
- `mvn.μ`

# Deep kernel learning

- Deep kernel learning : ニューラルネットワークの結果をカーネルに入れてあとはカーネル法と同じ

$$\bullet k(x) = e^{-\frac{d(nn(x), nn(x'))^2}{2}} \text{ にする}$$

- あとはturingライブラリと違うabstractgpsとfluxで計算する行が可能になる

- カーネルは前と同じtransformの部分を関数にする

- `k = SqExponentialKernel() ∘ FunctionTransform(neuralnet)`

- あとはライブラリの公式のままにする

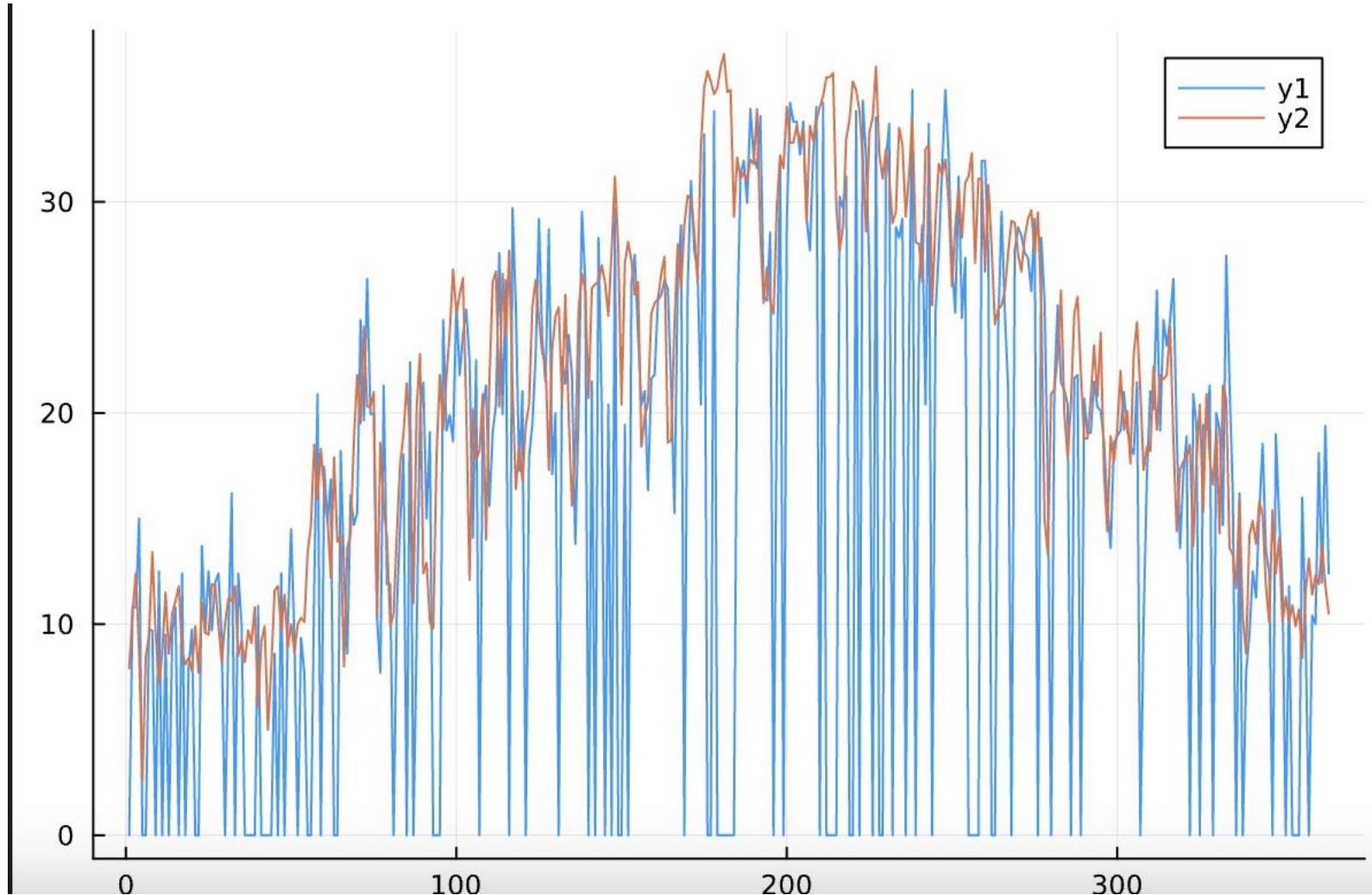
- この部分はコードでわかりやすいのでコードで表す

□ — ⊢

- neuralnet = Chain(Dense(1, 20), Dense(20,30),Dense(30, 1))
- k = SqExponentialKernel() ∘ FunctionTransform(neuralnet) nmax = 1000
- gpprior = GP(k) noise\_std = 0.01
- x\_train=collect(eachrow(vec(x1))) ,
- y\_train = vec(y1).+rand(364)\*noise\_std
- y\_train = vec(y1).+rand(364)\*noise\_std
- fx = AbstractGPs.FiniteGP(gpprior, x\_train, 0.01^2)
- fp = posterior(fx, y\_train) ; ps = Flux.params(k)
- loss(y) = -logpdf(fx, y), opt = Flux.ADAM(0.1)
- for i in 1:nmax
- grads = gradient(ps) do
- loss(y\_train)
- end
- Flux.Optimise.update!(opt, ps, grads)
- end
- pred = marginals(posterior(fx, vec(y1))(ColVecs(x2)))
- .

# 結果

- $(150.8812989137516, 0.8419378752168479)$

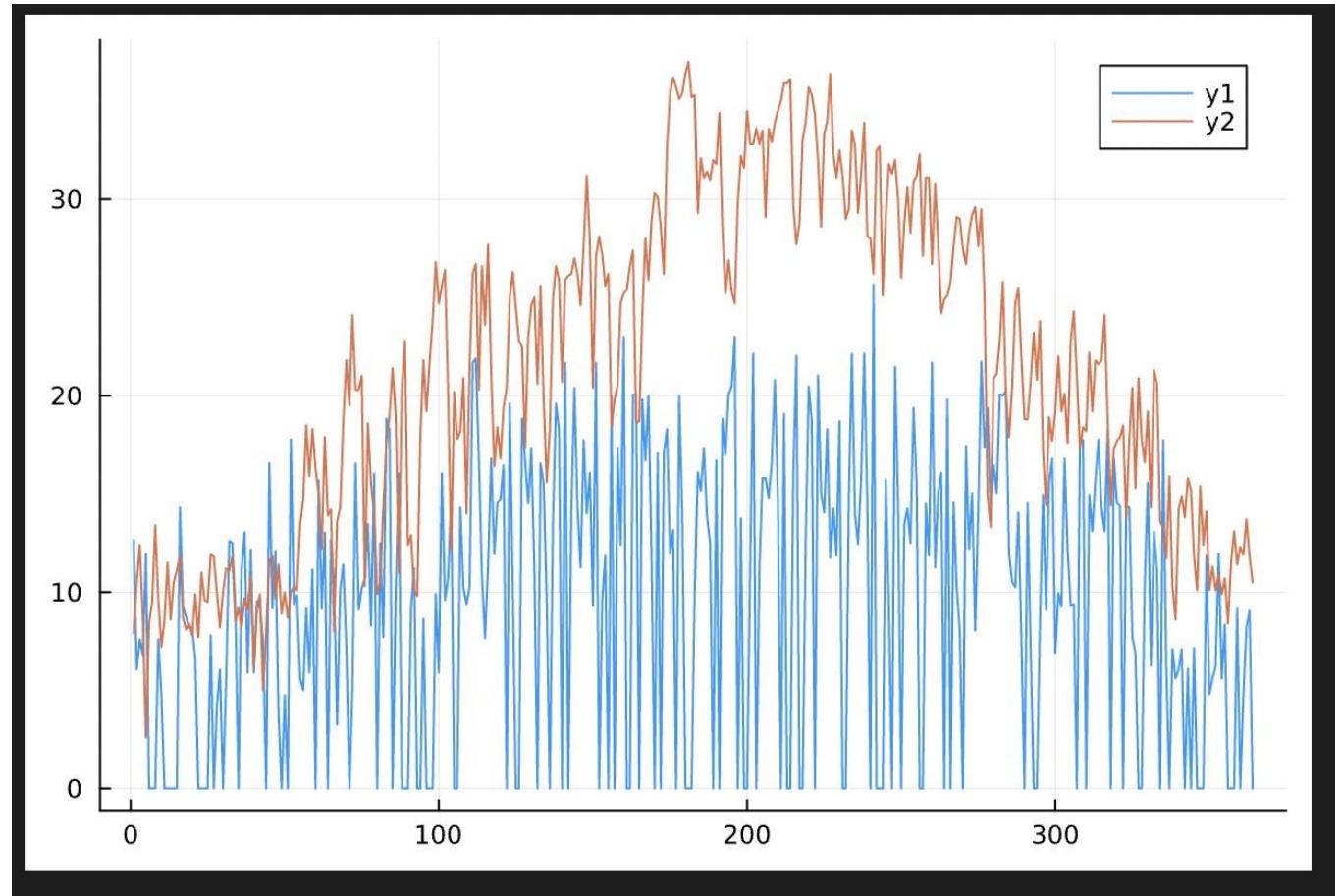


# Turingでやってみる

- 前の方法の操作簡単は簡単が結果あまり良くない前にやったガウス過程などに踏まえてturingでやってみよう
- ガウス過程と同じベイズとガウス推論を使う
- これらの方針を結合するとturing のサンプリング時間がすごく長くなる(500回のサンプリングが110分間ぐらいかかった)

```
@model function DKN(x,y,np,re,alpha=0.09)
    • noise = 1e-6
    • α ~ MvLogNormal(MvNormal(zeros(np), I/alpha))
    • nn=re(α)
    • #σ ~ Normal(0.0, 1.0)
    • k=GP(sekernel(nn))
    • kk=k(ColVecs(x),noise)
    • y ~ MvNormal(mean(kk), (cov(kk)+noise*I))
    • return nothing
    • end;
```

# 結果



# 確率方法のまとめ

- ガウス過程は有効だが計算が面倒臭い 実際の結果も良くない
- Deep kernelにすると 平均の場合は時々 0 が出る
- ベイジアンニューラルネットワークの効果も余良くない
- ネットの資料では確率方法に関連する例が少ないため今回はどこかが間違っているかもしれない
- Turing を使う場合は最後線形代数の計算になった
- Turingは個人的におすすめしないライブラリー できるだけそのまま計算できるライブラリーの方がいい

# 全体のまとめ

- 全体的に言うとどの方法の結果も良くない 失敗したといえる
- これまでの失敗を全部見ると気温の変化は前の日の気温とノイズがある微分方程式で表せるかもしれないそのノイズは色々ものに関連するので確率分布で表示するのがいいかもしれない

# 今回の課題について

- ・元々はコードのファイルもアップロードしたいが10個ぐらいのコードのファイルの中は全部こちやこちやになっていて整理が難しいのでやめたその代わりにスライドにペストした
- ・以前機械学習やニューラルネットワークの本質は線形代数と聞いたがあまり実感しなかった。今回は確かに線形代数が重要だと感じた。確率や線形代数に勉強不足のため最後の確率方法はあまりうまくいかなかった。

# Juliaについて

- 個人的な感想だがjuliaは確かに効率が高いかつ簡単に書けると思う juliaを使うにはプログラミングの知識より計算科学と数学の知識が重要だと思う。
- Juliaの一番よく使うライブラリーはLinearAlgebraだと思う。