Apresentação - Redes Neurais com Flux em Julia

Gabriel Vinicius Ferreira

Junho de 2023

Sumário

1 Introdução

A apresentação elaborada junto com esse roteiro busca explicar de maneira simples o conceito básico do desenvolvimento de uma rede neural em Julia utilizando o Flux. O modelo da rede tratada aqui segue a estrutura de uma rede de perceptron, utilizada para a classificação de dados de maneira supervisionada.

Acesso ao GitHub:
`https://github.com/gabriel-ferr/perceptron-seno`

1.1 O que é um perceptron?

Um perceptron é uma estrutura simples composta por uma matriz de entrada, uma matriz de pesos, uma função de combinação e uma função de ativação

não linear resultando em uma saída. Essa estrutura é construída de maneira semelhante a figura 1.

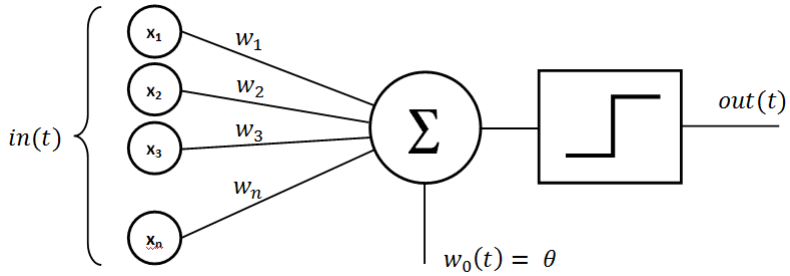

Figura 1: Estrutura básica de um perceptron.

Podemos escrever essa estrutura pela relação matemática da equação (1), onde W_0 é nomeado BIAS do neurônio e φ a função de ativação não linear.

$$y(X) = \varphi(W_0 + \sum_{i=1}^{n} W_i X_i) \tag{1}$$

Um perceptron sozinho não tem muita utilidade prática, então, normalmente trabalhamos com redes de perceptrons.

1.2 Perceptron de múltiplas camadas

Para tornar a estrutura do perceptron de fato útil, nós encadeamos várias camadas dessa estrutura a fim de obter um melhor resultado. Para isso, passamos as saídas de uma camada como entrada na camada seguinte, como na figura 2.

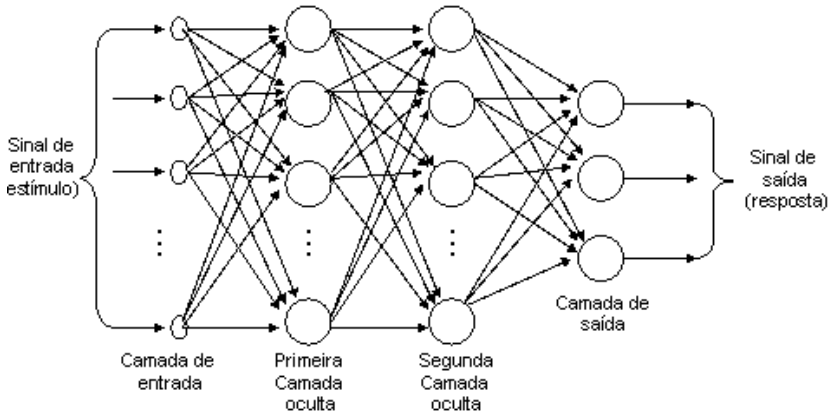

Figura 2: Estrutura perceptron de múltiplas camadas.

Dessa maneira, buscamos armazenar informação relacionada aos resultados corretos nos pesos, então, para fazer com que essa rede funcione, precisamos fazer o ajuste dos pesos.

1.2.1 Treinamento de uma rede de múltiplas camadas

Vou tratar do treinamento supervisionado, em que os resultados são comparados com os valores desejados e o erro é utilizado para calcular um ajuste para cada peso da rede.

$$\Delta W_{j,i}(t) = \eta \delta_j(t) y_i(t) \tag{2}$$

A maneira como esse ajuste é calculado é chamado de função de otimização, um exemplo simples dessa função optimização seria a equação (2), o gradiente local do neurônio que recebe a informação ($\delta_j(t)$) pode ser calculado pelas equações (3) e (4), onde $W_{k,j}$ é o peso entre o neurônio atual e um na camada seguinte e $v_j(t)$ o valor que entrou no neurônio, e o valor de $y_i(t)$ representa a saída do neurônio na camada anterior, antes de ser mutliplicado pelo peso $W_{j,i}$.

$$\delta_(t) = erro_j(t)\varphi'_j(v_j(t)) \tag{3}$$

$$\delta_(t) = \varphi'_j(v_j(t)) \sum_n \delta_k(t) W_{k,j}(t) \tag{4}$$

O erro presente na equação (3) é calculado por uma função de perda que recebe a saída da rede (no caso, $y_j(t)$) e o valor esperado de resposta, calculado uma taxa de erro que será aplicada na correção dos pesos.

2 Conjunto de dados usados

n	ω
1	0.411
2	5.566
3	1.624
4	0.213
5	0.074
6	3.561
7	1.222
8	2.667
9	0.316
10	0.251

Tabela 1: Velocidade angular (ω) utilizada nos testes, os valores foram gerados a partir de uma distribuição normal.

Para as simulações feitas, segui a ideia proposta pelo Professor Tiago. Para isso, gerei um conjunto de 10 velocidades angulares (ω) a partir de uma distribuição normal, como na tabela 1. Também gerei um conjunto de dados para servir como uma espécie de ruído nas curvas, esse valor será atribuído como a fase (ϕ) na equação (5). Foram gerados 20 valores para ϕ, 10 usando uma distribuição normal e 10 sendo valores totalmente aleatórios, registrados na tabela 2.

n	ϕ_{randn}	ϕ_{rand}
1	2.108	-1.258
2	3.051	0.612
3	0.438	2.511
4	2.812	-3.571
5	0.810	5.122
6	0.579	0.023
7	-2.830	1.113
8	-4.701	-0.782
9	0.865	2.367
10	0.357	3.157

Tabela 2: Fases usadas como ruído para a função seno (5). ϕ_{randn} refere-se aos dados baseados em uma distribuição normal e ϕ_{rand} dados inseridos aleatoriamente, sem nenhuma conexão.

$$y(t) = sin(\omega t + \phi) \tag{5}$$

Em seguida, gerei um conjunto aleatório de 1000 entradas no intervalo entre

0 e 10, atribuidos a uma variável t e inseridos na equação (5), com isso, temos um conjunto de mil pontos para cada uma das fases associadas a uma velocidade angular. Isso representa cerca de 20 mil pontos para cada velocidade angular que usaremos para o treinamento e teste da rede.

Para gerar esses dados utilizei o código da figura 3, como falta-me prática na manipulação de matrizes com o Julia, utilizei uma maneira mais bruta de fazer a construção da matriz de dados, utilizando um conjunto de `for`, porém, certamente há maneiras mais práticas de fazer isso.

Com isso, o objetivo da rede será, dados os valores de $y(t)$ gerados pela equação (5), classificar qual das 10 velocidades angular deu origem ao conjunto de dados. Para isso, a entrada da rede deve ser o conjunto de mil pontos relacionados a uma determinada velocidade e fase, e a saída será um conjunto de 10 valores de probabilidade, cada neurônio representando uma posição de 1 a 10 da tabela 1.

```
1   using Flux, Plots, Statistics, ProgressMeter
2
3   ω_list = [0.411, 5.566, 1.624, 0.213, 0.074, 3.561, 1.222, 2.667, 0.316, 0
4   ϕ_list = [2.108, 3.051, 0.438, 2.812, 0.810, 0.579, -2.830, -4.701, 0.865,
5
6   times = range(0, stop=10, length=1000)
7
8   data = Matrix{Float32}(undef, 200, 1000)
9
10  for i = 1:10
11      for j = 1:20
12          for p = 1:1000
13              ω = ω_list[i]
14              ϕ = ϕ_list[j]
15              local t = times[p]
16              data[(i - 1) * 10 + j, x] = sin(ω * t + ϕ)
17          end
18      end
19  end
```

Figura 3: Código que faz a construção da matriz de dados utilizada para os testes e para o treino da rede neural.

3 Criando o modelo da rede com o Flux

Para facilitar, peguei alguns códigos prontos da documentação do Flux para essa sessão.

```
function linear(in, out)
  W = randn(out, in)
  b = randn(out)
  x -> W * x .+ b
end

linear1 = linear(5, 3) # we can access linear1.W etc
linear2 = linear(3, 2)

model(x) = linear2(σ.(linear1(x)))

model(rand(5)) # => 2-element vector
```

Figura 4: código adquirido a partir da documentação do Flux[2], nele temos a construção aberta de um modelo de rede neural com perceptrons.

A maneira mais detalhada que o Flux permite para a construção é o código da figura 4. Nele, temos de maneira aberta como a função de combinação é montada, no caso sendo linear, e o encademaneto das camadas com o uso de uma função de ativação não linear.

```
layer1 = Dense(10 => 5, σ)
# ...
model(x) = layer3(layer2(layer1(x)))
```

Figura 5: código adquirido a partir da documentação do Flux[2], ele simplifica a estrutura construída no código da figura 4.

```
using Flux

layers = [Dense(10 => 5, σ), Dense(5 => 2), softmax]

model(x) = foldl((x, m) -> m(x), layers, init = x)

model(rand(10)) # => 2-element vector
```

Figura 6: código adquirido a partir da documentação do Flux[2], ele simplifica a estrutura construída no código da figura 4.

Apesar da construção ser relativamente simples em relação ao mesmo código em C, C++ ou C#, ele ainda não faz uso das facilidades do Flux, já que a biblioteca permite simplificar o código usando modelos prontos. Como no caso das figuras 5 e 6, onde a função de combinação da figura 4, `function linear(in, out)`,

foi substituída pela função `Dense(in => out, func)` previamente preparada pelo Flux.

```
model2 = Chain(
  Dense(10 => 5, σ),
  Dense(5 => 2),
  softmax)

model2(rand(10)) # => 2-element vector
```

Figura 7: código adquirido a partir da documentação do Flux[2], ele simplifica completamente a construção do modelo da rede, inicialmente proposto por 4.

Por fim, podemos simplificar o encadeamento da rede usando a estrutura `Chain(...)` fornecida pelo Flux, os parametros passados por essa estrutura são as funções que compõem a rede, podendo ela pertencer ao Flux ou ser criada pelo usuário conforme o modelo da documentação, isso porque algumas estruturas precisam de características especiais para serem reconhecidas pelas funções de treinamento da biblioteca. Um exemplo do uso dessa estrutura de encadeamento é o código da figura 7, além disso, já que não vou aprofundar a construção dessas funções, vou utilizar um código semelhante ao da figura para a rede utilizada nesse projeto.

Um ponto interessante a ser comentado é que o Flux permite a construção simples de modelos não lineares de neurônios, onde a informação e divida e volta a se unir em algum ponto da rede antes de ser exibido o resultado final. Apesar de ser interessante, não irei trabalhar com isso aqui, porém, achei válido comentar sobre a capacidade disponibilizada pela biblioteca.

3.1 Modelo utilizado no projeto

Como mencionado, para esse projeto utilizei uma proposta de modelo semelhante ao da figura 7, alterando o número de neurônios e as funções utilizadas.

Não realizei testes de performace com variações de modelo para buscar qual seria melhor para o problema proposto, apenas criei um modelo que me pareceu que iria funcionar.

Como demonstrado na figura 8, cada camada da rede sofre uma redução de neurônios, reduzindo o conjunto de mil pontos de entrada, características do conjunto de dados, para dez valores em uma função softmax.

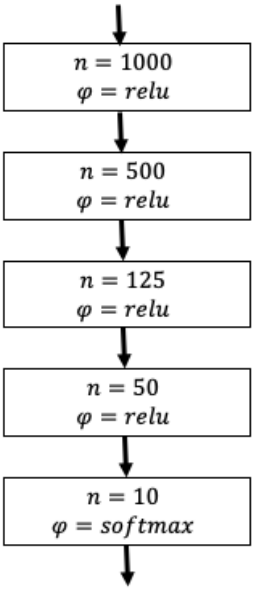

Figura 8: Modelo da rede utilizado no projeto.

```
105    model = Chain(
106        Dense(1000, 500, relu),
107        Dense(500, 125, relu),
108        Dense(125, 50, relu),
109        Dense(50, 10),
110        softmax
111    )
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Progress: 100%|
Chain(
  Dense(1000 => 500, relu),             # 500_500 parameters
  Dense(500 => 125, relu),              # 62_625 parameters
  Dense(125 => 50, relu),               # 6_300 parameters
  Dense(50 => 10),                      # 510 parameters
  NNlib.softmax,
)                   # Total: 8 arrays, 569_935 parameters, 2.175 MiB.
```

Figura 9: código que estrutura a rede neural proposta pela figura 8. Ele segue a mesma lógica do código indicado na figura 7

O código da figura 9 é responsável por criar a estrutura da rede proposta na figura 8, sendo exibido na saída da criação do modelo o número de parâmetros e o espaço utilizado na memória pela rede. É importate notar que esse espaço é a alocação dos parametros da rede, não envolvendo os dados de treinamento.

Caso entremos com os dados gerados anteriormente nesse modelo, vamos

obter como saída um conjunto de dados de probabilidade fornecidos pela função softmax, representada pela equação (6). Assim, a probabilidade fornecida pelo neurônio 1 seria a probabilidade da velocidade angular utilizada para gerar os dados for a 1, e a mesma lógica aplicando-se às 10 velocidades utilizadas.

```
10×200 Matrix{Float32}:
 0.101192   0.098345   0.158601   0.097925
 0.0878354  0.0826372  0.0773421  0.086412
 0.0878354  0.0826372  0.0773421  0.086412
 0.0878354  0.0826372  0.0836591  0.086412
 0.0878354  0.0826372  0.0773421  0.086412
 0.121717   0.160115   0.0773421  0.150008
 0.0878354  0.0826372  0.0773421  0.086412
 0.0878354  0.0826372  0.0950444  0.086412
 0.138178   0.0850244  0.0896039  0.0921529
 0.111901   0.160693   0.186381   0.141443
```

Figura 10: resultado obtido ao inserir o conjunto de dados na rede não treinada.

$$\phi(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{n} e^{z_k}} \tag{6}$$

O resultado, quando impresso, é semelhante a figura 10, onde cada linha representa a probabilidade conforme explicado anteriormente, e cada coluna um dos dados de teste. Em seguida, criei uma função para retornar o número de referência da velocidade angular conforme a tabela 1, essa função pode ser observada no código da figura 11, também foi feito de maneira improvisada, porém, o objetivo que seria retornar um número de 1 a 10 referente a maior probabilidade foi satisfeito.

```
function get_graphdata_from_output(output)
    result = []
    for i = 1:200
        maior = 1
        if (output[2, i] > output[maior, i]) maior = 2 end
        if (output[3, i] > output[maior, i]) maior = 3 end
        if (output[4, i] > output[maior, i]) maior = 4 end
        if (output[5, i] > output[maior, i]) maior = 5 end
        if (output[6, i] > output[maior, i]) maior = 6 end
        if (output[7, i] > output[maior, i]) maior = 7 end
        if (output[8, i] > output[maior, i]) maior = 8 end
        if (output[9, i] > output[maior, i]) maior = 9 end
        if (output[10, i] > output[maior, i]) maior = 10 end
        push!(result, maior)
    end
    return result
end
```

Figura 11: função responsável por retornar o resultado a partir da probabilidade de cada saída.

Caso coloquemos isso em um gráfico em que o eixo x recebe a contagem de 1 a 200 referente as colunas da matriz de cado (como temos 10 velocidades angular e 20 fases, temos 200 funções seno distintas) e o eixo y recebe o número de referência para cada velocidade angular (de 1 a 10), o desejado seria obter uma espécie de "gráfico de escada", subindo 1 "degrau"a cada 20 unidades de x. Fazendo isso com os dados obtidos da saída a partir da rede sem treinamento, teremos um gráfico semelhante a figura 12.

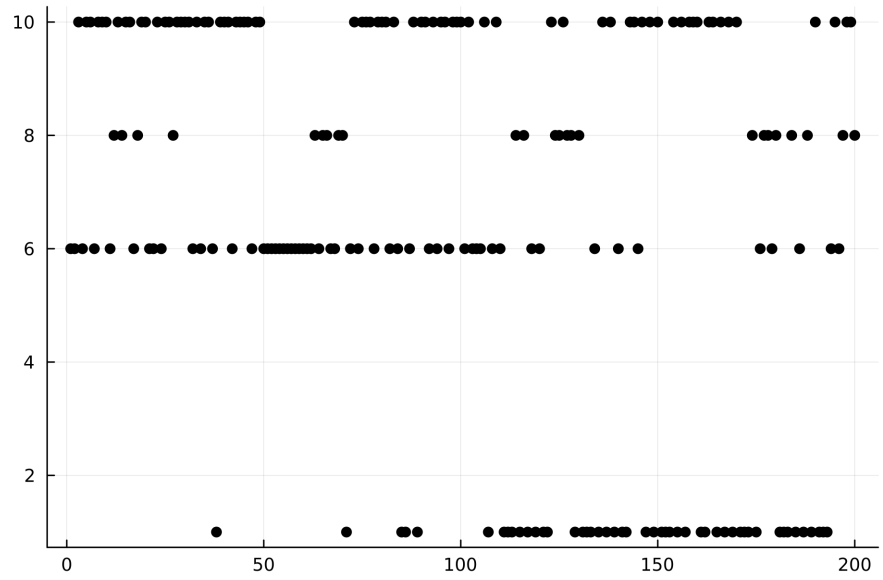

Figura 12: gráfico referente ao resultado da classificação ao inserir o conjunto de dados na rede não treinada.

4 Treinamento da rede

O treinamento da rede é o que de fato faz tudo acontecer. O ideal seria possuir um conjunto de dados para treinamento e outro conjunto para testes, porém, estarei usando o mesmo conjunto para as duas atividades, o que pode resultar em um erro de precisão não registrado ao tentar usar um outro conjunto posteriormente. Por isso, talvez seja recomendado tentar variar o máximo possível o conjunto de dados enquanto se tenta modelar uma rede neural para alguma atividade.

Com o Flux o treinamento da rede é relativamente simples, a biblioteca fornece uma função única que faz todo o treinamento em uma única linha, porém, não acho muito interessante o seu uso já que ela limita muito o controle dessa etapa tão importante.

```
38    desired_values = get_desired_values()
39    loader = Flux.DataLoader((data', desired_values), batchsize = 64, shuffle = true)
```

Figura 13: código responsável por gerar uma matriz de valores desejados e de carregar os dados no Flux.

Para fazer o treinamento manual, precisamos seguir quatro etapas básicas, a primeira é carregar o conjunto de dados que vamos utilizar para o treinamento. Isso pode ser observado sendo feito no código da figura 13. A função `get_desired_values()` retorna uma matriz semelhante a da figura 10 (uma matriz 10x100), porém, com a diferença de que cada coluna é preenchida com zeros, com exceção da linha referente a velocidade angular que recebe o valor um referenciando a velocidade angular correta (basicamente, a cada 20 o um desce uma linha, não sei se foi uma explicação muito válida).

A função `Flux.DataLoader()` é uma ferramenta disponibilizada pelo Flux para o carregamento desses dados. Primeiro passamos como parâmetro o par (x, y) onde x é a matriz de entrada e y a saída desejada. Uma coisa importante é que o número de linhas na matriz de entrada deve ser igual ao número de neurônios e entrada, e que as matrizes passadas para x e y devem possuir o mesmo número de colunas. Isso ocorre devido ao modo como o Flux funciona internamente, então, usamos o operador `'` para transpor a matriz de dados gerada anteriormente. O `batchsize` representa o tamanho dos blocos de dados que vamos passar, e o `shuffle` informa que queremos embaralhar o nosso conjunto de dados, isso é importante para garantir que a rede aprenda de maneira correta e não fique travada a um grupo de dados.

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

Figura 14: definição da função de optimização utilizada.

Em seguida, devemos definir a função de otimização que iremos utilizar, para isso, seguimos o código da figura 14. O Flux disponibiliza diversas funções previamente preparadas que podem ser utilizadas, no caso, estarei utilizando a função `Adam` com uma taxa de aprendizagem de 0.01, é possível optar por outas funções disponibilizadas na documentação do Flux, em
`https://fluxml.ai/Flux.jl/stable/training/optimisers/`.

```
42    loss(y_hat,y) = Flux.crossentropy(y_hat, y)
```

Figura 15: definição da função de perda utilizada.

Para finalizar a configuração, devemos definir uma função de perda, essa função pode ser uma das fornecidas pelo Flux em
`https://fluxml.ai/Flux.jl/stable/models/losses/` ou criada como função

qualquer no Julia. No caso, estarei utilizando a função Cross Entropy fornecida pelo Flux como função de perda, como demonstrado na figura 15. Tentei utilizar outras funções de perda, porém, o resultado não foi muito bom.

```
122  losses = []
123  @showprogress for epochs = 1:200
124      for (x, y) in loader
125          loss_r, grads = Flux.withgradient(model) do m
126              y_hat = m(x)
127              loss(y_hat, y)
128          end
129          Flux.update!(optim, model, grads[1])
130          push!(losses, loss_r)
131      end
132  end
```

Figura 16: definição da estrutura que realiza o treinamento da rede.

Por fim, criamos a estrutura de treinamento para a rede, como na figura 16. Para isso, determinamos um número de épocas em que o treinamento será realizado, no caso 200 tempos, pegamos cada elemento carregado na figura 13 e passamos para a rede, calculando a perda e os gradientes e fazemos a atualização dos valores de peso da rede com base nesses valores e na função de otimização adotada.

5 Testando e resultados obtidos

Para os testes fiz uma série de tentativas rodando a rede para 375, 400 e 425 épocas, em seguida escolhi dois resultados dentre as execuções realizadas (cerca de 10 para cada), as duas opções escolhidas foram aquelas que aparentavam ter um melhor resultado dentre as alternativas.

Para 375 épocas, podemos observar os resultados obtidos nas figuras 17 e 18. Para o resultado 17 é possível perceber o início da formação das linhas em formato de "escada"desejado (já que o esperado seria a cada 20 valores no eixo horizontal deslocar 1 unidade para cima no eixo vertical), além disso, o erro registrado está bem grande. Para a figura 18 o resultado é até pior, com um erro maior.

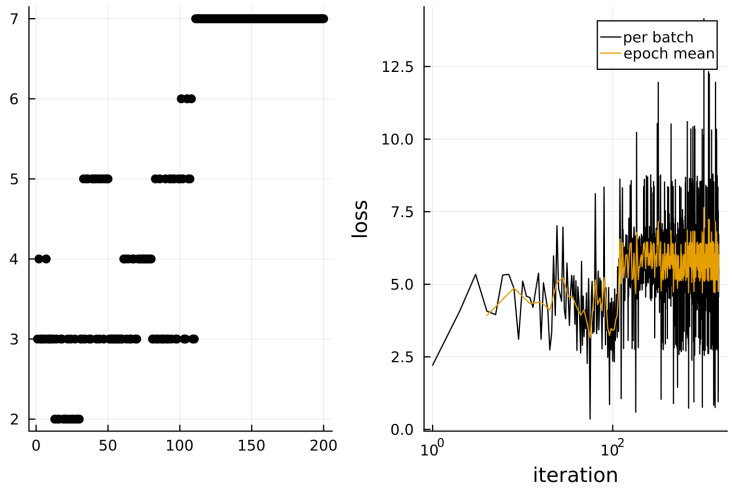

Figura 17: gráfico representando um dos testes realizados com 375 épocas de treinamento.

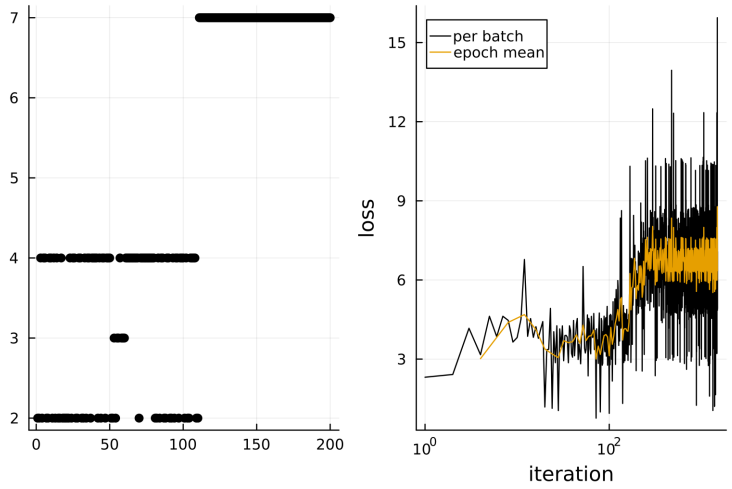

Figura 18: gráfico representando um dos testes realizados com 375 épocas de treinamento.

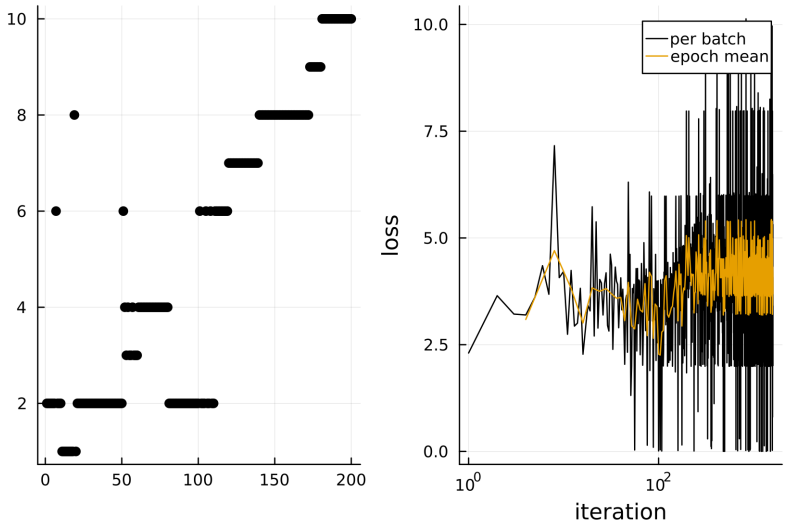

Figura 19: gráfico representando um dos testes realizados com 400 épocas de treinamento.

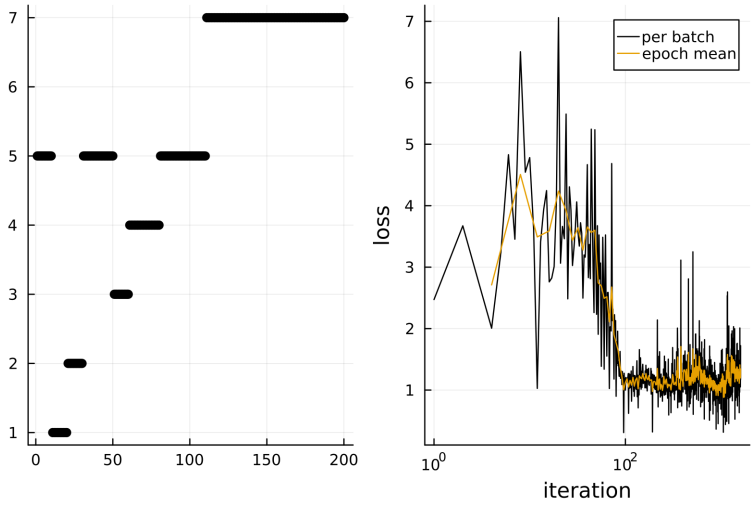

Figura 20: gráfico representando um dos testes realizados com 400 épocas de treinamento.

Aumentando o número de épocas de 375 para 400, podemos verificar um resultado muito melhor nas figuras 19 e 20, apesar de ainda apresentarem diversos pontos fora do desejado e um erro expressivo.

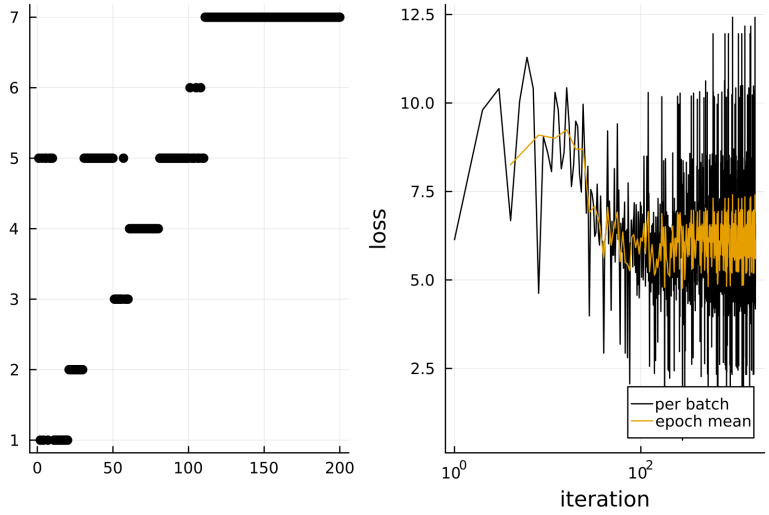

Figura 21: gráfico representando um dos testes realizados com 425 épocas de treinamento.

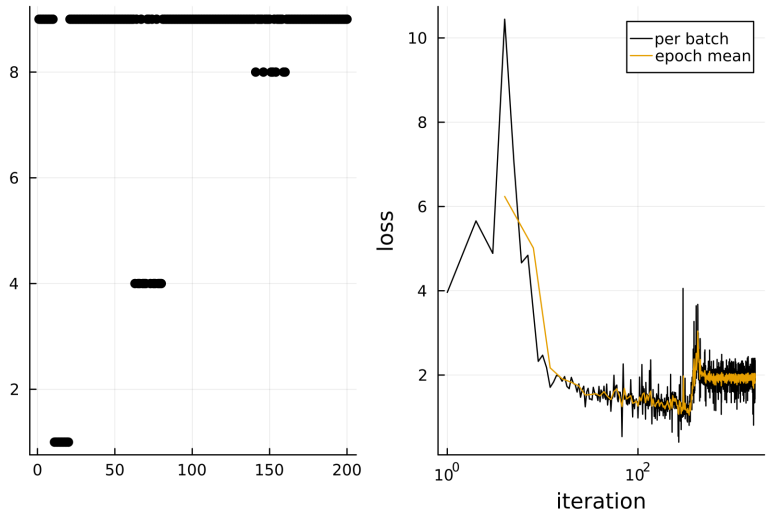

Figura 22: gráfico representando um dos testes realizados com 425 épocas de treinamento.

Aumentar o número de épocas de treinamento pode parecer uma ideia interessante para melhorar a performace, porém, não necessariamente isso irá funcionar, já que pode tornar a rede tendenciosa. Isso pode ser observado nos resultados das figuras 21 e 22, onde, na figura 21 o resultado foi algo semelhante ao esperado, porém, com diversos erros, e na figura 22 observamos que, apesar do baixo erro resgistrado, o resultado está muito longe do esperado.

Como aumentar o número de épocas não melhorou a qualidade da rede, optei por remover o conjunto de dados gerados sem nenhuma relação. Em outras palavras, removi a metade dos dados onde o valor da fase foi gerada de maneira totalmente aleatória, mantendo a parte retirada de uma distribuição normal. Uma outra opção seria normalizar todos os dados.

Repetindo o teste com 375 épocas, já é possível observar uma melhora gigantesca na qualidade dos resultados, obtendo baixos valores de perda e um gráfico semelhante ao esperado, com poucos erros, como podemos observar nas figuras 23 e 24 .

Aumentado o número de épocas, obtemos o resultado das figuras 25 e 26, onde é possível perceber uma melhora significativa, com um erro ainda menor e obtendo alguns resultados que seguem exatamente o esperado para o conjunto utilizado. Por fim, o mesmo se repeta para as análises com 425 épocas, como pode ser observado nas figuras 27 e 28, sem muitas alterações quando comparamos com 400 épocas.

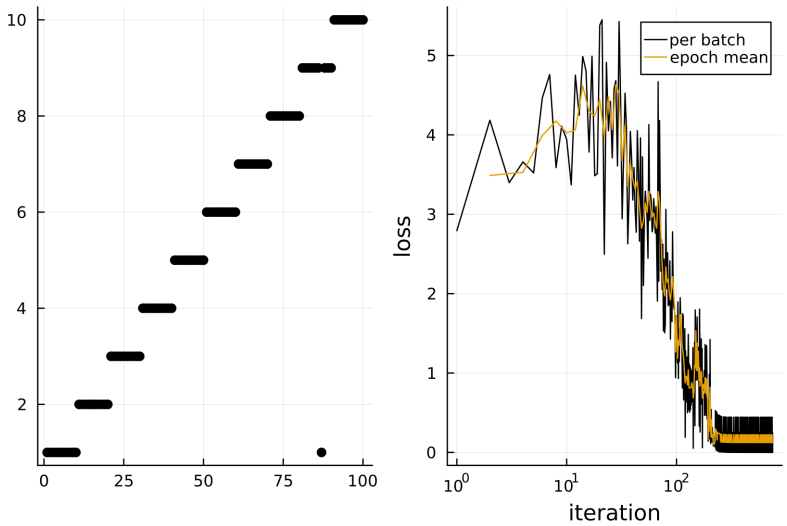

Figura 23: gráfico representando um dos testes realizados com 375 épocas de treinamento apenas com os dados gerados com ϕ_{randn} da tabela 2.

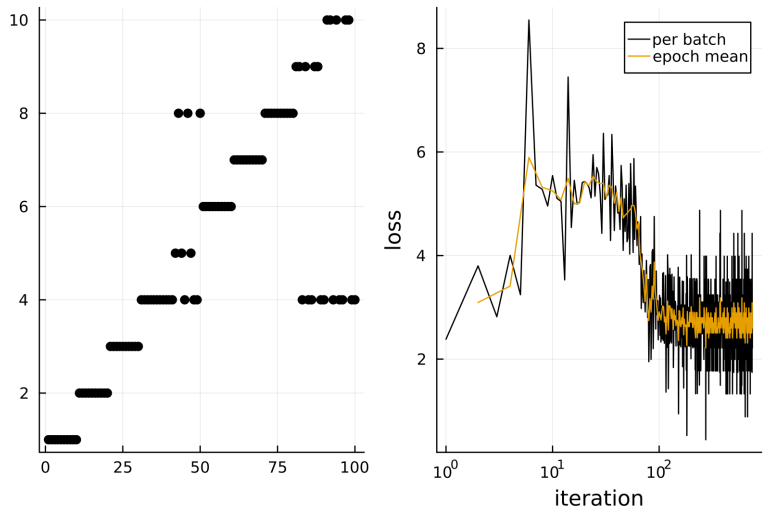

Figura 24: gráfico representando um dos testes realizados com 375 épocas de treinamento apenas com os dados gerados com ϕ_{randn} da tabela 2.

Assim, pontuo uma coisa que deve ser feita antes de inserir as características de um sistema na rede neural, essas características devem estar associadas por uma relação normalizada com média zero. Como os dados utilizados foram gerados com essa intenção, foi possível separá-los e realizar os testes novamente, porém, o ideal é realizar a normalização de todos os conjuntos de dados antes de inserir eles na rede.

O livro do Haykin cita uma série de fatores para melhorar o desempenho de um algoritmo de retropropagação, que é o modelo utilizado aqui, que incluem essa normalização dos dados de entrada e uma atribuição que deve ser feita à taxa de aprendizagem para que a rede aprenda de maneira uniforme em todas as camadas. (Redes Neurais: princípios e prática, página 205 à 211 [1])

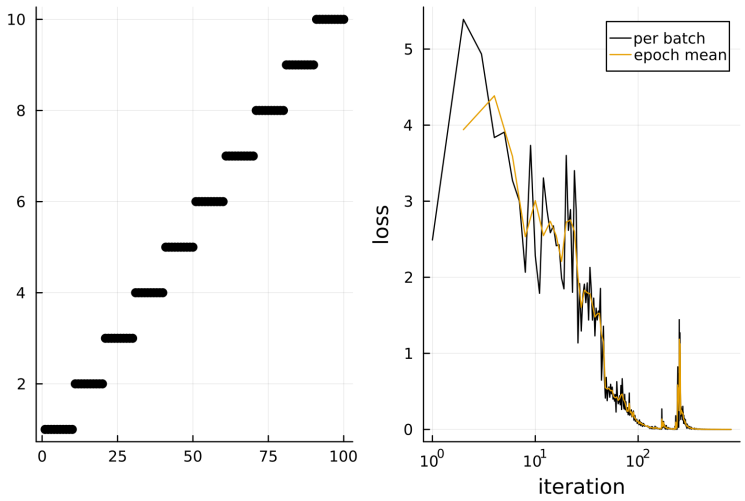

Figura 25: gráfico representando um dos testes realizados com 400 épocas de treinamento apenas com os dados gerados com ϕ_{randn} da tabela 2.

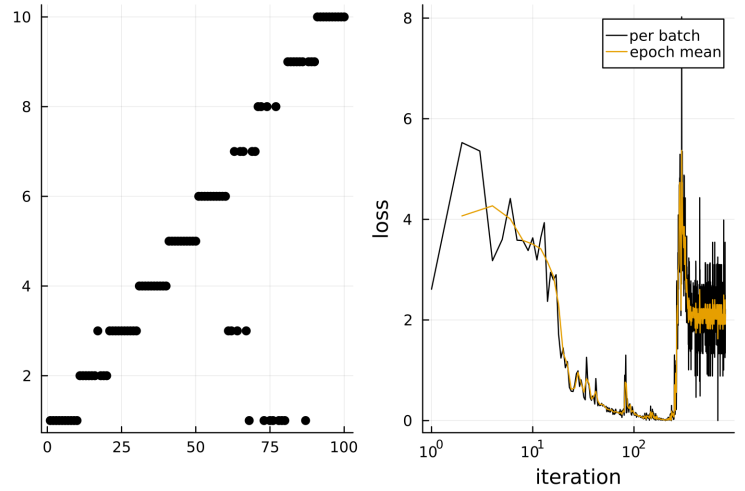

Figura 26: gráfico representando um dos testes realizados com 400 épocas de treinamento apenas com os dados gerados com ϕ_{randn} da tabela 2.

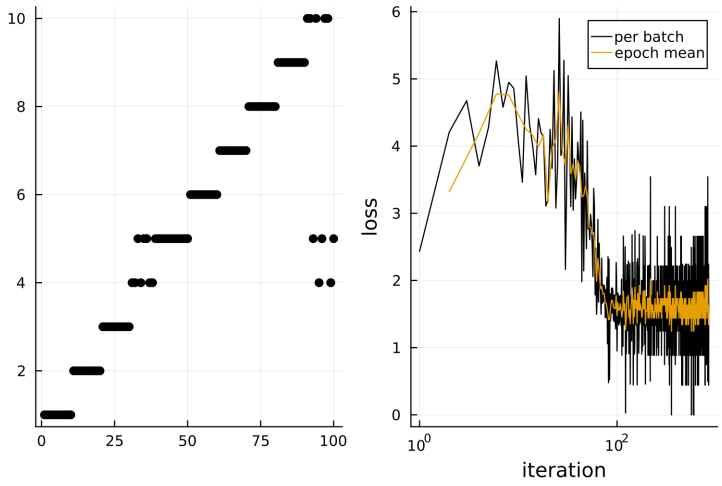

Figura 27: gráfico representando um dos testes realizados com 425 épocas de treinamento apenas com os dados gerados com ϕ_{randn} da tabela 2.

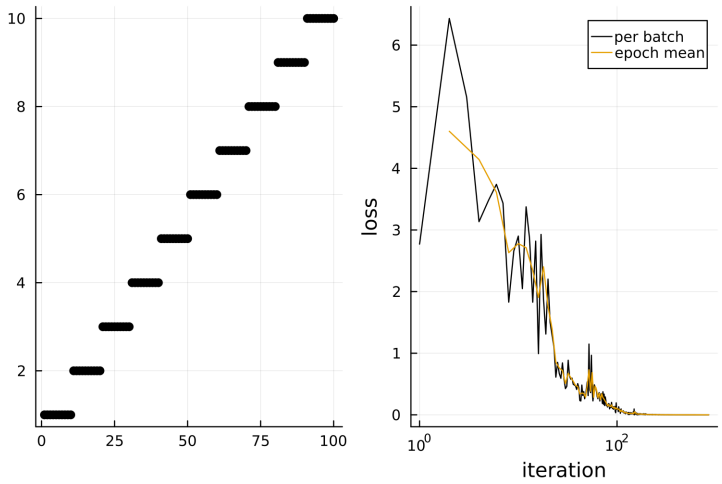

Figura 28: gráfico representando um dos testes realizados com 425 épocas de treinamento apenas com os dados gerados com ϕ_{randn} da tabela 2.

6 Utilizando GPU

Por fim, a parte mais esperado, o uso da GPU com o Flux. A documentação do Flux[2] disponibiliza uma descrição bem completa sobre o uso de GPU com a biblioteca, infelizmente não consegui testar devido aos computadores a minha

disposição não serem compatíveis (Macbook com placa de vídeo integrada no processador).

Apesar disso, a figura 29 demonstra um exemplo simples de como funciona o uso da GPU com o Flux em Julia, não tenho certeza se funciona corretamente, porém, fica exemplificado que, mesmo quando utilizado sem compatibilidade, o tratamento de erros do Julia e da biblioteca garantem o funcionamento do código sem muitos problemas.

```
105    model = Chain(
106        Dense(1000, 500, relu),
107        Dense(500, 125, relu),
108        Dense(125, 50, relu),
109        Dense(50, 10),
110        softmax
111    ) |> gpu

117    loader = Flux.DataLoader((data', desired_values), batchsize = 64, shuffle = true) |> gpu

134    out_2 = get_graphdata_from_output(model(data'|>gpu)|>cpu)
135    graph_2 = scatter(x[1:100], out_2[1:100], legend=false)

[ Info: The GPU function is being called but the GPU is not accessible.
[ Defaulting back to the CPU. (No action is required if you want to run on the CPU).
Progress: 100%|

julia>
```

Figura 29: exemplo de código para o uso da GPU com o Flux em Julia.

7

Bibliografia

[1] Simon Haykin. *Redes Neurais: princípios e prática, 2-ed. ISBN 978-85-7307-718-6*. Bookman, 2001.

[2] https://fluxml.ai/Flux.jl/stable/.