# Exemplo – UDE (dinâmica toda)

Laura Costa Pereira Miranda – lauram@lncc.br

31 de maio de 2021

# A EDO

Queremos treinar uma rede neural afim de aproximar a solução do seguinte tipo de equação diferencial ordinária:

$$\begin{cases} u'(t) = & f(u(t), t), \quad \forall t \in (t_0, T] \\ u(t_0) = & u_0 \end{cases}$$

# A EDO

- Redes neurais são conhecidas como aproximadores universais.
- Se $NN(u,t) \approx u'(u,t)$, então $NN(u,t) \approx f(u(t),t)$.

# A EDO

- Podemos definir a seguinte função custo para o problema de otimização envolvendo a rede neural:

$$L = \left( \sum_{i}^{n} (NN(u(t_i), t_i) - f(u(t_i), t_i))^2 + \right.$$
$$\left. + \left(g(NN(u(t_i), t_1)) - u(t_i))^2\right) \right)^{\frac{1}{2}}$$

- com $g(t)$, sendo a solução para $NN(u(t_i), t_i)$ utilizando o método de Runge-Kutta de 4ª ordem.

# Exemplo

O problema é o mesmo mostrado anteriormente:

$$\begin{cases} u'(t) = & f(u(t), t) \ = \ 2t, \quad \forall t \in (0, 1] \\ u(t_0) = & 1 \end{cases}$$

cuja solução analítica é dada por:

$$u(t) = \int f(t)dt + C,$$

de forma que $u(t) = t^2 + 1$.

# Parâmetros utilizados

- Taxa de aprendizado (gradiente descendente estocástico): $1 \cdot 10^{-2}$.
- Passos de aprendizado (épocas): 100 / 200.
- Neurônios de entrada: 1.
- Neurônios de saída: 1.
- Métodos de otimização: Gradiente Descendente Estocástico / ADAM.

# Parâmetros utilizados

- Funções de ativação: tangente hiperbólica / sigmóide / relu.
- Número de camadas ocultas: 2 / 1.
- Neurônios por camada oculta: 32 / 64.
- Pontos de treinamento: 10 / 20.

# As redes neurais

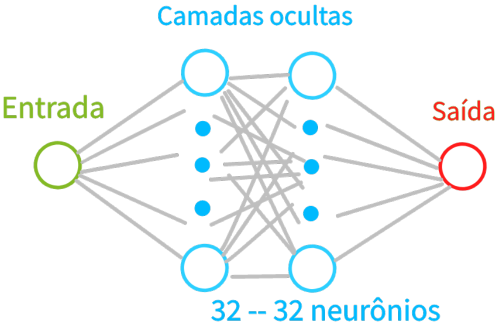



Figura: Ilustração da rede de 2 camadas e 32 neurônios utilizada nos experimentos

# As redes neurais

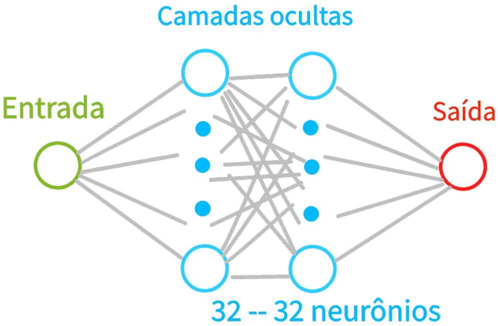



Figura: Ilustração da rede de 1 camada e 32 neurônios utilizada nos experimentos

# As redes neurais

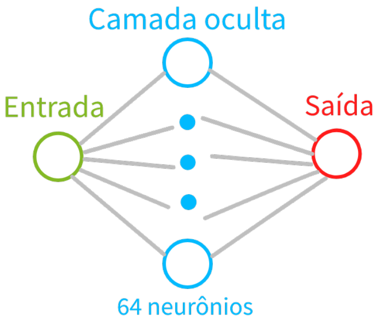



Figura: Ilustração da rede de 1 camada e 64 neurônios utilizada nos experimentos

# As redes neurais

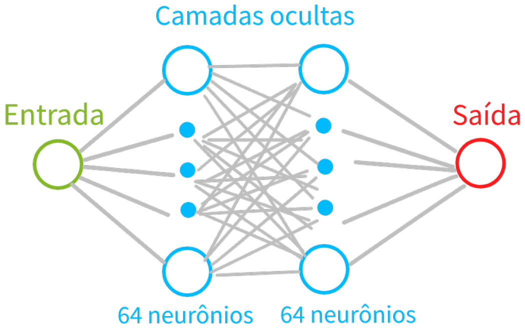



Figura: Ilustração da rede de 2 camadas e 64 neurônios utilizada nos experimentos

# Funções de ativação utilizadas

$$\boxed{\begin{array}{lcl} f(x) = \tanh(x) & - & \text{Tangente Hiperbólica} \\ f(x) = \dfrac{1}{1 + e^( - x)} & - & \text{Sigmóide} \end{array}}$$

# Modificações no código

```
dNN =  self._multilayer_perceptron(x)
```

```
dNN =  self._multilayer_perceptron(x) #(self._g(x + self.inf_s) - self._g(x))/sel
summation.append((dNN-self.fx_vec[i])**2.0 + (self._g(x) - self.sx_vec[i])**2.0)
```

```
# Fixed Seed?
if self.seed is not None:
        tf.random.set_seed(self.seed)
```

```
self.sx_vec = self._true_solution(self.x_vec)
```

```
result.append(self._g(val).numpy()[0][0][0][0][0][0][0][0][0][0][0][0])
```

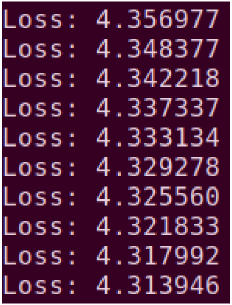
Figura: Modificações com respeito à rede, função custo, semente e dados de treinamento e resultado, respectivamente

# Modificações no código

```python
def _g(self, x):
        u0 = self._f0()
        if(x==0.0):
                u0 = tf.constant([[[[[[[[[[[[u0]]]]]]]]]]]], dtype='float32')
                return u0
        else:
                dx = 1e-2
                k = 0.0
                while(k<x):
                        #print("while",x,k,u0)
                        k1 = self._multilayer_perceptron(u0)
                        k2 = self._multilayer_perceptron(u0+dx*k1*0.5)
                        k3 = self._multilayer_perceptron(u0+dx*k2*0.5)
                        k4 = self._multilayer_perceptron(u0+dx*k3)
                        u1 = u0+(dx/6.0)*(k1+2.0*k2+2.0*k3+k4)
                        u0 = float(u1)
                        k = k+dx
                        #print("while2",k,u0)
                return u1
```

Figura: Modificações com respeito à g

# Resultados



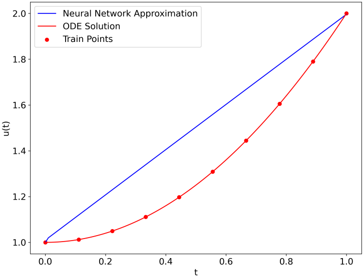
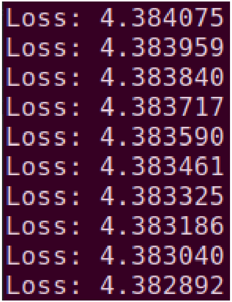




Figura:

# Resultados



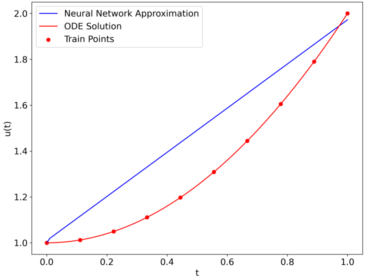
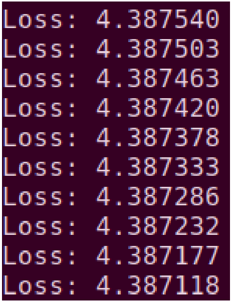




Figura:

# Resultados



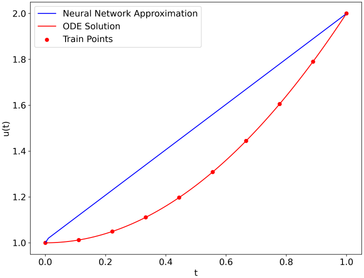




Figura:

# Resultados

```
Loss: 4.053600
Loss: 2.458205
Loss: 2.402673
Loss: 2.390347
Loss: 2.388959
Loss: 2.385156
Loss: 2.382922
Loss: 2.381095
Loss: 2.379286
Loss: 2.378026
```

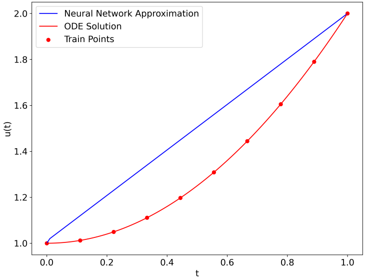
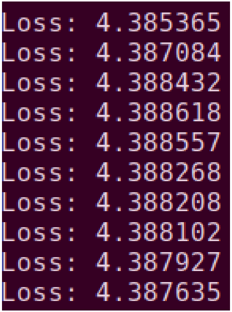



Figura:

# Resultados


Loss: 4.385365
Loss: 4.387084
Loss: 4.388432
Loss: 4.388618
Loss: 4.388557
Loss: 4.388268
Loss: 4.388208
Loss: 4.388102
Loss: 4.387927
Loss: 4.387635

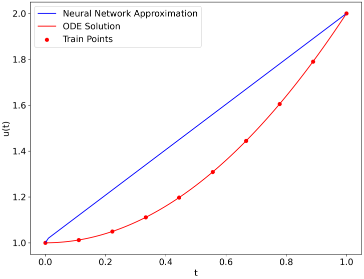
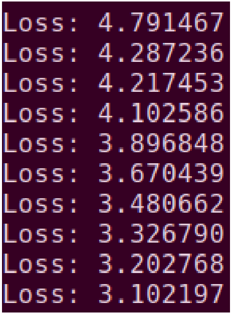



Figura:

# Resultados



Loss: 4.791467
Loss: 4.287236
Loss: 4.217453
Loss: 4.102586
Loss: 3.896848
Loss: 3.670439
Loss: 3.480662
Loss: 3.326790
Loss: 3.202768
Loss: 3.102197

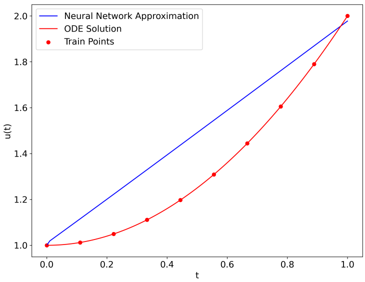



Figura:

# Resultados

```
Loss: 4.385365
Loss: 4.387084
Loss: 4.388432
Loss: 4.388618
Loss: 4.388557
Loss: 4.388268
Loss: 4.388208
Loss: 4.388102
Loss: 4.387927
Loss: 4.387635
```

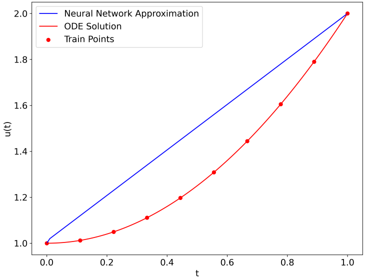
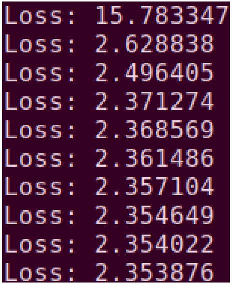



Figura:

# Resultados


```
Loss: 15.783347
Loss: 2.628838
Loss: 2.496405
Loss: 2.371274
Loss: 2.368569
Loss: 2.361486
Loss: 2.357104
Loss: 2.354649
Loss: 2.354022
Loss: 2.353876
```

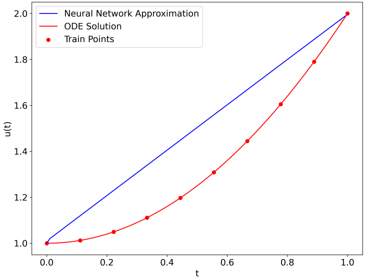
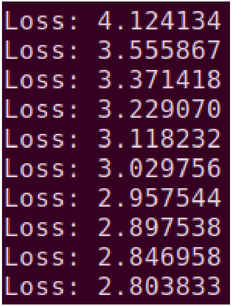



Figura:

# Resultados



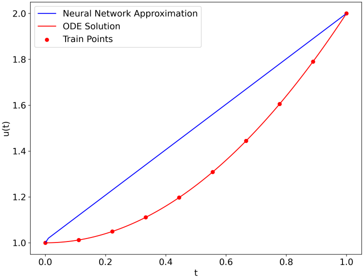




Figura:

# Resultados

```
Loss: 6.998922
Loss: 5.852427
Loss: 5.349732
Loss: 5.074384
Loss: 4.905894
Loss: 4.795350
Loss: 4.718608
Loss: 4.662805
Loss: 4.620706
Loss: 4.588020
```

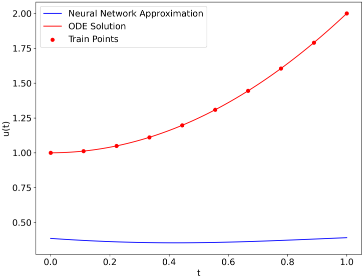
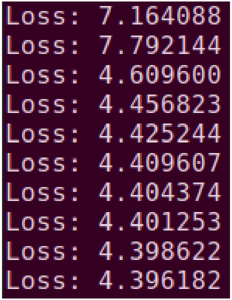



Figura:

# Resultados



Loss: 7.164088
Loss: 7.792144
Loss: 4.609600
Loss: 4.456823
Loss: 4.425244
Loss: 4.409607
Loss: 4.404374
Loss: 4.401253
Loss: 4.398622
Loss: 4.396182

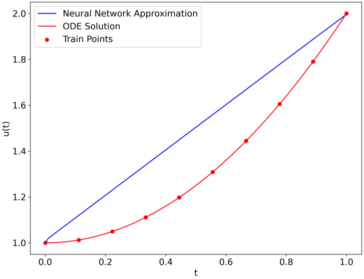



Figura:

# Referências


Lagaris, I. E.; Likas, A.; FOTIADIS, D. I.; **Aritificial Neural Networks for Solving Ordinary and Partial Differential Equations**. Berlin: Springer, 2010.

RESENDE, A. C. M.; **Neural Networks and Deep Learning for Dynamical Systems**.