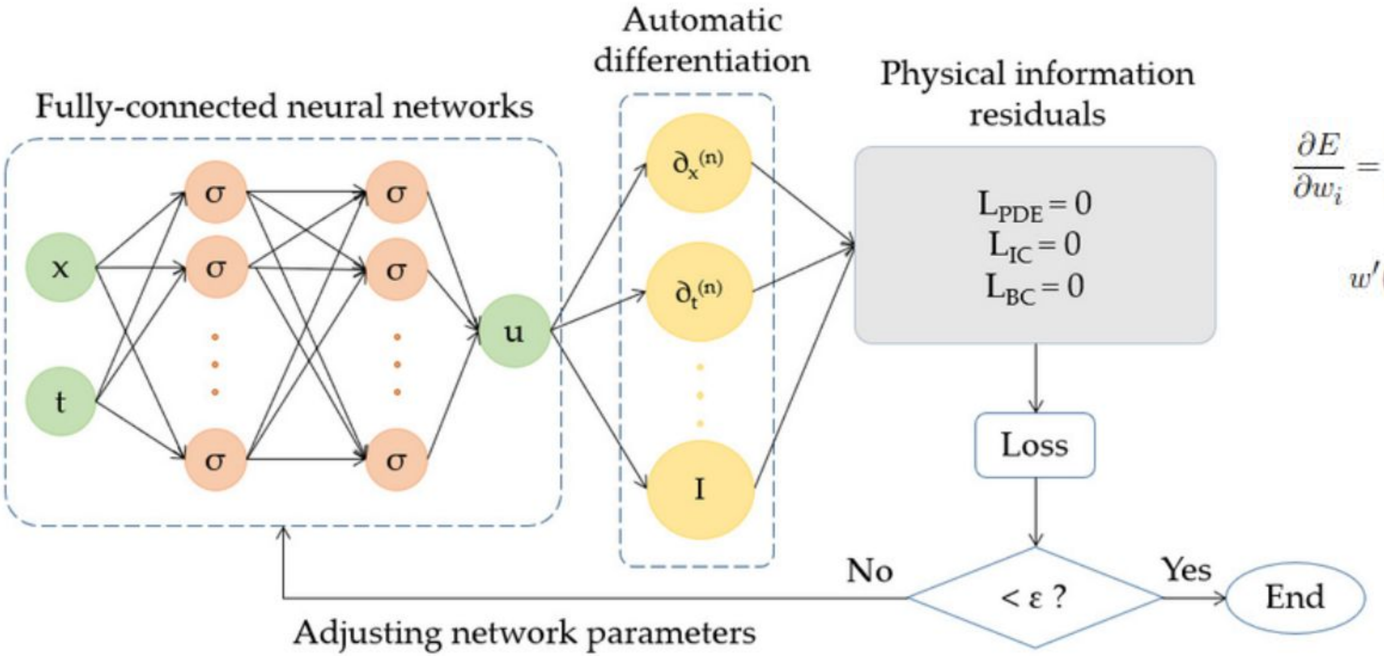
Apresentação - PINNS

Introdução PINNs

- As saídas das redes neurais MLP podem ser derivadas em relação a qualquer uma das entradas usando a Diferenciação automática (que é o cálculo automático do gradiente descendente).
- Com isso, é possível inserir o custo da EDP no treinamento, fazendo com que a rede neural aprenda a física no processo.

Arquitetura PINNs



$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial O}\frac{\partial O}{\partial w_i} = 2(O_p - O_r)\sigma'(w_i)$$

$$w'(m+1) = w'(m) - \alpha\frac{\partial E}{\partial w'}$$

Caso da equação de calor 1D

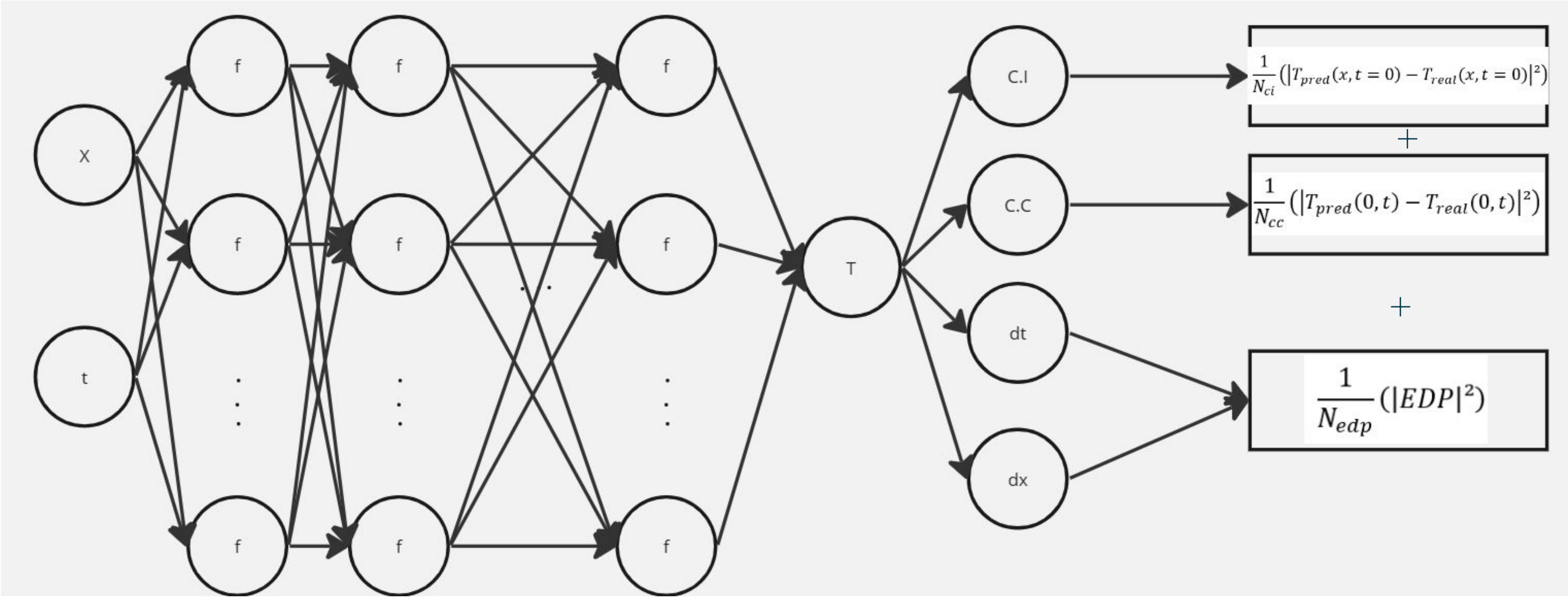
Layer (type)	Output Shape	Param #
lambda (Lambda)	(None, 2)	0
dense (Dense)	(None, 20)	60
dense_1 (Dense)	(None, 20)	420
dense_2 (Dense)	(None, 20)	420
dense_3 (Dense)	(None, 20)	420
dense_4 (Dense)	(None, 20)	420
dense_5 (Dense)	(None, 20)	420
dense_6 (Dense)	(None, 20)	420
dense_7 (Dense)	(None, 20)	420
dense_8 (Dense)	(None, 1)	21

$$\frac{\partial T}{\partial t} - c^2 \frac{\partial^2 T}{\partial x^2} = 0$$

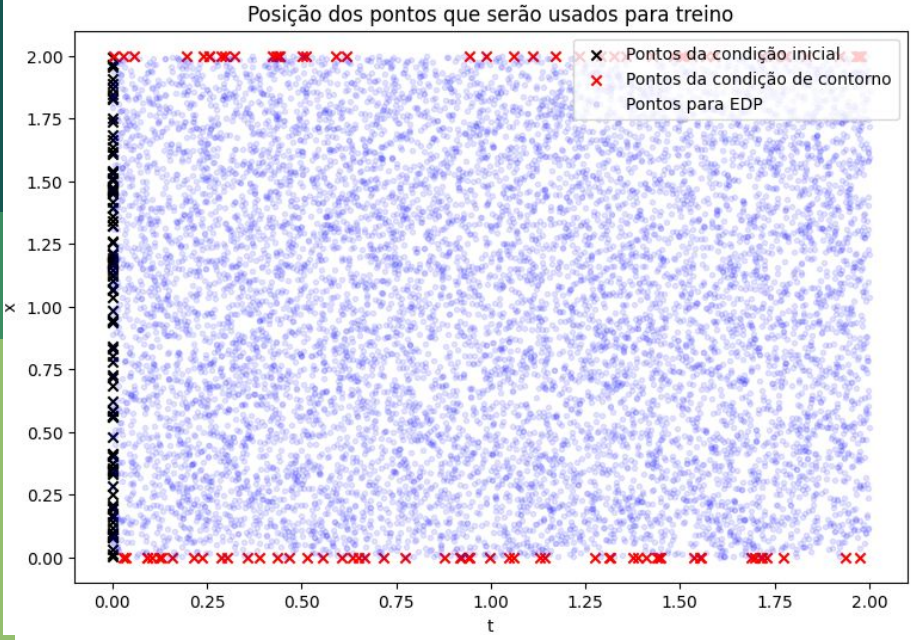
$$T(t=0, x) = x^2(2-x)$$

$$T(t, x=0) = T(t, x=2)$$

Caso da equação de calor 1D



Caso da equação de calor 1D




```python
def inicial(x):
  return (x**2)*(2-x)

def contorno(t, x):
  n = x.shape[0]
  return tf.zeros((n,1))

#Definindo os pontos X
N_0 = 100 #100 pontos para condição inicial
N_b = 100 #100 pontos para condição de contorno
N_r = 10000 #Pontos para a edp

#Pontos do domínio, dado pelo Maziar
tmin = 0. ; tmax = 2.
xmin = 0.; xmax = 2.

#Ponto inferior e superior, respectivamente
lb = tf.constant([tmin, xmin]); ub = tf.constant([tmax, xmax])

#Obtendo pontos para a condição inicial
t0 = tf.zeros((N_0,1))*lb[0]
x0 = tf.random.uniform((N_0,1), lb[1], ub[1]) #Colocando os valores de x0 em ordem aleat
x0 = tf.concat([t0, x0], 1) #Criando uma matriz com os valores de tempo = 0 e de x0

#Valores de u para a condição inicial
u_ini = inicial(x0[:,1:2])

#Repetindo o processo, mas para a condição de contorno
tb = tf.random.uniform((N_b,1), lb[0], ub[0])
xb = lb[1] + (ub[1] - lb[1]) * keras.backend.random_bernoulli((N_b,1), 0.5)
xb = tf.concat([tb, xb], 1)

#Valores na condição de contorno
u_cont = contorno(xb[:,0:1], xb[:,1:2])

#Repetindo o processo, mas agora é para obter os pontos da EDP
tr = tf.random.uniform((N_r,1), lb[0], ub[0])
xr = tf.random.uniform((N_r,1), lb[1], ub[1])
xr = tf.concat([tr, xr], 1)
```

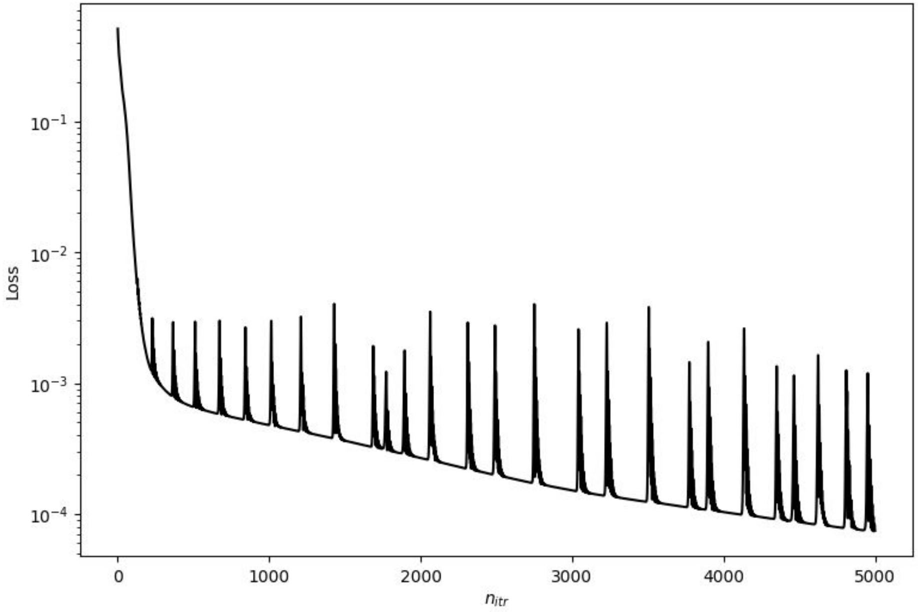
Caso da equação de calor 1D

```python
def gradiente(modelo, X_r):
  with tf.GradientTape(persistent=True) as tape:

    #Registrando tempo e posição para a diferenciação automática
    t, x = X_r[:, 0:1], X_r[:,1:2]
    tape.watch(t)
    tape.watch(x)

    #previsão do modelo
    u = modelo(tf.stack([t[:,0], x[:,0]], 1))
    #gradiente du/dx
    ux = tape.gradient(u, x)


    #gradiente du/dt
  ut = tape.gradient(u, t)
  #gradiente du²/d²x
  uxx = tape.gradient(ux, x)

  del tape

  return ut -difus *uxx
```

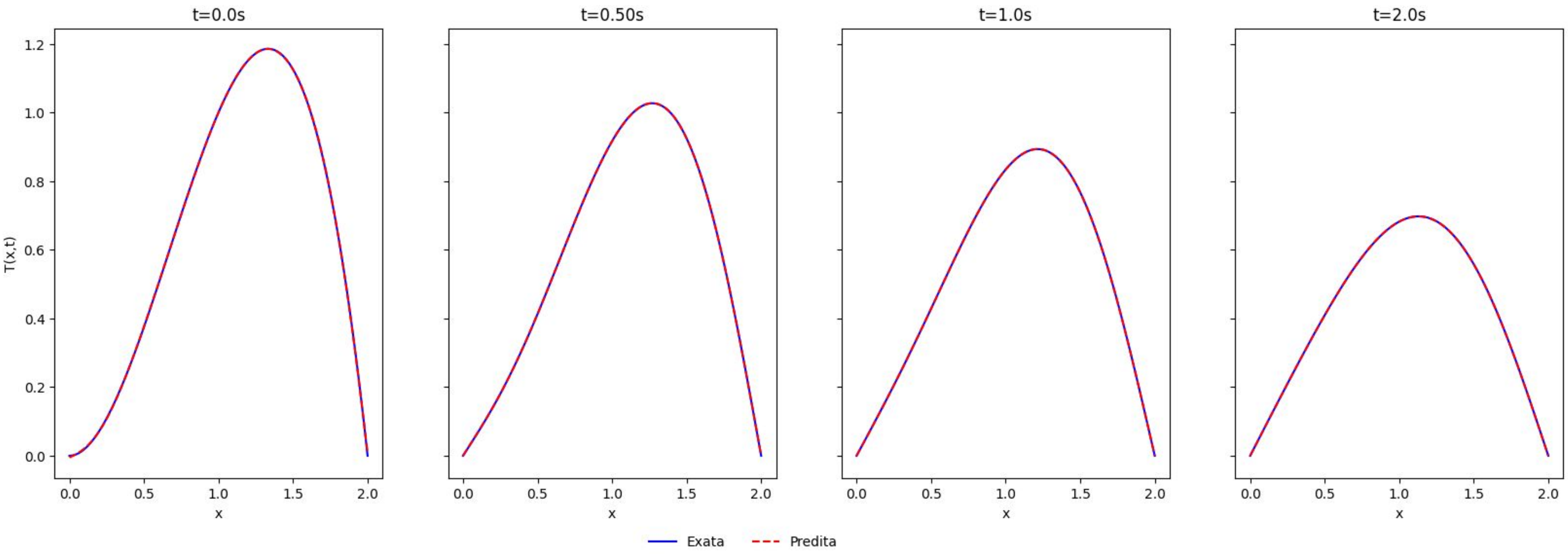
```python
def MSE(modelo, xr, X_cond, u_cond):

    #Erro edp
    r = gradiente(modelo, xr)
    erro = tf.reduce_mean(tf.square(r))

    loss = erro

    #Erro da rede neural
    for i in range(len(X_cond)):
        u_pred = modelo(X_cond[i])
        loss += tf.reduce_mean(tf.square(u_cond[i] - u_pred))

    return erro, loss

def grad(modelo, xr, X_cond, u_cond):
  #tirando o gradiente em relação ao modelos, para que eles seja
  with tf.GradientTape(persistent=True) as tape:
    tape.watch(modelo.trainable_variables)
    erro, loss = MSE(modelo, xr, X_cond, u_cond)

  g = tape.gradient(loss, modelo.trainable_variables)
  del tape

  return erro, loss, g
```

Caso da equação de calor 1D

```
#etapa de treinamento como uma função do TensorFlow para aumentar a velocidade do treinamento
@tf.function
def train_step(modelo):
  #Calculando a perda do modelo em relação ao modelo, com a função grad
  erro, loss, grad_theta = grad(modelo, xr, X_cond, u_cond)

  #Aplicando o gradiente as variaveis do modelo de rede neural
  otimizador.apply_gradients(zip(grad_theta, modelo.trainable_variables))

  return erro, loss

itr = 5000
historico = []
erro_aux = []
t0 = time()

for i in range(itr+1):

    erro, loss = train_step(modelo)

    #Salvando os erros para listar
    historico.append(loss.numpy())
    erro_aux.append(erro.numpy)

    if i%10 == 0:
        print(i,"Loss treino: {:10.8e}, Loss edp: {:10.8e}".format(loss, erro))

print('\nTempo de treino da rede neural: {} segundos'.format(time()-t0))
```



Resultado do modelo para a equação do calor 1D



Caso de Buckley-Leverett

$$\frac{\partial Sw}{\partial t} + \frac{\partial f(Sw)}{\partial Sw}\frac{\partial Sw}{\partial x} = 0$$

$$f(Sw) = \frac{Sw^2}{Sw^2 + \frac{(1 - Sw)^2}{M}}$$

A equação envolvendo a permeabilidade relativa e viscosidade sempre ia para **not a number** no treinamento da rede, optei por usar essa equação

$$Sw(x, t = 0) = 0$$

$$Sw(x = 0, t) = 1$$

$$Sw(x = 1, t) = 0$$

Caso de Buckley-Leverett

```
def gradiente(modelo, X_r):
  with tf.GradientTape(persistent=True) as tape:
    t, x = X_r[:, 0:1], X_r[:,1:2]
    tape.watch(t)
    tape.watch(x)
    u = modelo(tf.stack([t[:,0], x[:,0]], 1))
    tape.watch(u)

    k_rw = ((u - Swi)/(1-Swi-Swo))**2
    k_ro = ((1- u - Swi)/(1-Swi-Swo))**2
    #f = (k_rw/u_w)/((k_rw/u_w) + (k_ro/u_o))
    #f = 1/(1+(k_ro/u_w)*(u_o/k_rw))
    f = fw(u, 2)
    f_u = tape.gradient(f,u)
    f_x = tape.gradient(f,x)

    ux = tape.gradient(u, x)
    ut = tape.gradient(u, t)
    uxx = tape.gradient(ux, x)

  del tape
```

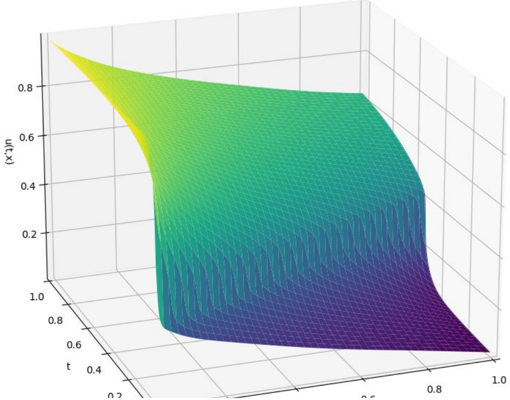
```
#Definindo os pontos X
N_0 = 300 #100 pontos para condição inicial
N_b = 300 #100 pontos para condição de contorno
N_r = 23000 #Pontos para a edp

#Pontos do domínio
tmin = 0. ; tmax = 1.
xmin = 0.; xmax = 1.

#Ponto inferior e superior, respectivamente
lb = tf.constant([tmin, xmin]); ub = tf.constant([tmax, xmax])

#Obtendo pontos para a condição inicial
t0 = tf.zeros((N_0,1))*lb[0]
x0 = tf.random.uniform((N_0,1), lb[1], ub[1]) #Colocando os valores de x0 em
x0 = tf.concat([t0, x0], 1) #Criando uma matriz com os valores de tempo = 0 e

#Valores de u para a condição inicial
u_ini = inicial(x0[:,1:2])

#Repetindo o processo, mas para a condição de contorno
tb = tf.random.uniform((N_b,1), lb[0], ub[0])
#xb = lb[1] + (ub[1] - lb[1]) * keras.backend.random_bernoulli((N_b,1), 0.5)
xb = tf.zeros((N_b,1))*lb[1]
xb = tf.concat([tb, xb], 1)

#Valores na condição de contorno
u_cont = contorno(xb[:,0:1], xb[:,1:2])

#Repetindo o processo, mas agora é para obter os pontos da EDP
tr = tf.random.uniform((N_r,1), lb[0], ub[0])
xr = tf.random.uniform((N_r,1), lb[1], ub[1])
xr = tf.concat([tr, xr], 1)

#Fazendo uma lista, para uso posterior
X_cond = [x0, xb]
u_cond = [u_ini, u_cont]
```

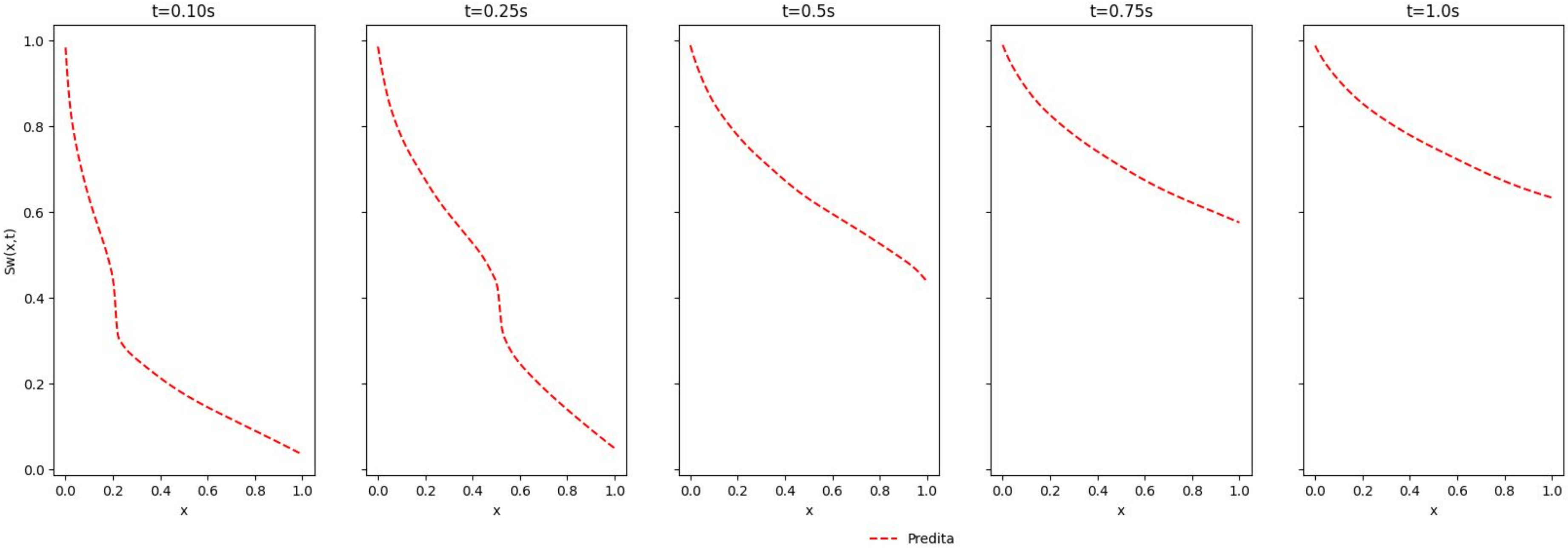
Caso de Buckley-Leverett

```python
def MSE(modelo, xr, X_cond, u_cond):

    #Erro edp
    r = gradiente(modelo, xr)
    erro = tf.reduce_mean(tf.square(r))

    loss = erro

    #Erro da rede neural
    for i in range(len(X_cond)):
        u_pred = modelo(X_cond[i])
        loss += tf.reduce_mean(tf.square(u_cond[i] - u_pred))

    return erro, loss

def grad(modelo, xr, X_cond, u_cond):
  with tf.GradientTape(persistent=True) as tape:
    tape.watch(modelo.trainable_variables)
    erro, loss = MSE(modelo, xr, X_cond, u_cond)

  g = tape.gradient(loss, modelo.trainable_variables)
  del tape

  return erro, loss, g
```

```python
@tf.function
def train_step(modelo):
  erro, loss, grad_theta = grad(modelo, xr, X_cond, u_cond)

  otimizador.apply_gradients(zip(grad_theta, modelo.trainable_variables))

  return erro, loss

itr = 5000
historico = []
erro_aux = []
t0 = time()

for i in range(itr+1):

    erro, loss = train_step(modelosc)

    #Salvando os erros para listar
    historico.append(loss.numpy())
    erro_aux.append(erro.numpy)

    if i%100 == 0:
        print(i,"Loss treino: {:10.8e}, Loss edp: {:10.8e}".format(loss, erro))

print('\nTempo de treino da rede neural: {} segundos'.format(time()-t0))
```

Caso de Buckley-Leverett





Caso de Buckley-Leverett
Otimizador L-BFGS-B

1. Projeção de gradiente;
2. Cálculo generalizado do ponto de Cauchy;
3. Minimização do subespaço;
4. Busca de linhas;
5. Aproximação hessiana de memória limitada.

$$m_k(x) = f(x_k) + \mathbf{g_k}^T(x - x_k) + \frac{1}{2}(x - x_k)^T \mathbf{B_k}(x - x_k)$$

$$P(t) = (x_k^0 - t\mathbf{g_k}), x_k \in [x_{min}, x_{max}]$$

Caso de Buckley-Leverett
Otimizador L-BFGS-B

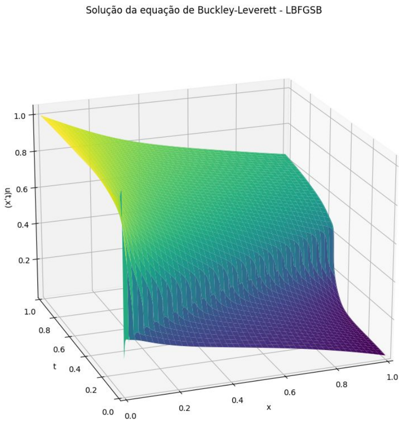
```python
def treino(self, X, u, method='L-BFGS-B', **kwargs):

    def get_weight_tensor():
        #Função para retornar variáveis atuais do modelo 

        weight_list = []
        shape_list = []

        #loop sobre todas as variáveis, ou seja, matrizes
        for v in self.modelo.variables:
            shape_list.append(v.shape)
            weight_list.extend(v.numpy().flatten())

        weight_list = tf.convert_to_tensor(weight_list)
        return weight_list, shape_list

    x0, shape_list = get_weight_tensor()
```

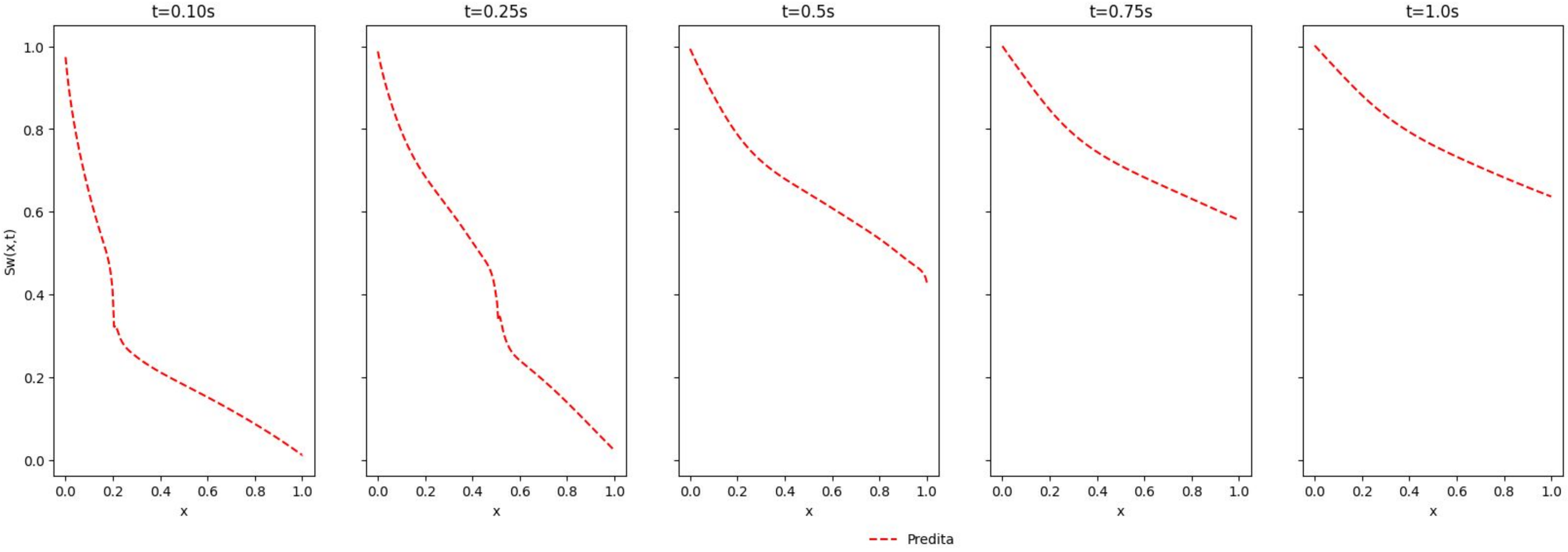
```python
def set_weight_tensor(weight_list):
    #Função que define lista de pesos para variáveis do modelo.
    idx = 0
    for v in self.modelo.variables:
        vs = v.shape

        #Matriz do peso
        if len(vs) == 2:
            sw = vs[0]*vs[1]
            new_val = tf.reshape(weight_list[idx:idx+sw],(vs[0],vs[1]))
            idx += sw

        #Vetor bias
        elif len(vs) == 1:
            new_val = weight_list[idx:idx+vs[0]]
            idx += vs[0]

        #Variáveis (no caso de configuração de identificação de parâmetr
        elif len(vs) == 0:
            new_val = weight_list[idx]
            idx += 1

        #Atribuir variáveis (cast necessário, pois o scipy requer o tipo
        v.assign(tf.cast(new_val, 'float32'))
```

Caso de Buckley-Leverett
Otimizador L-BFGS-B

```
def get_loss_and_grad(w):
    #Função que fornece perda de custo e gradiente em relação às variáveis treináveis como vetor.

    #Atualizar os pesos
    set_weight_tensor(w)
    #Determinar o custo da rede neural
    erro, loss, grad = self.grad(X, u)

    #Armazenando o custo atual para a função de retorno
    loss = loss.numpy().astype(np.float64)
    self.current_loss = loss

    #Salvando os valores do gradiente
    grad_flat = []
    for g in grad:
        grad_flat.extend(g.numpy().flatten())

    #Convertendo para array
    grad_flat = np.array(grad_flat,dtype=np.float64)

    #Retornando o custo e o gradiente
    return loss, grad_flat

#retorno, minimizando o gradiente
return scipy.optimize.minimize(fun=get_loss_and_grad,
                               x0=x0,
                               jac=True,
                               method=method,
                               callback=self.callback,
                               **kwargs)
```

Caso de Buckley-Leverett Otimizador L-BFGS-B





Reformulação da EDP de Buckley-Leverett

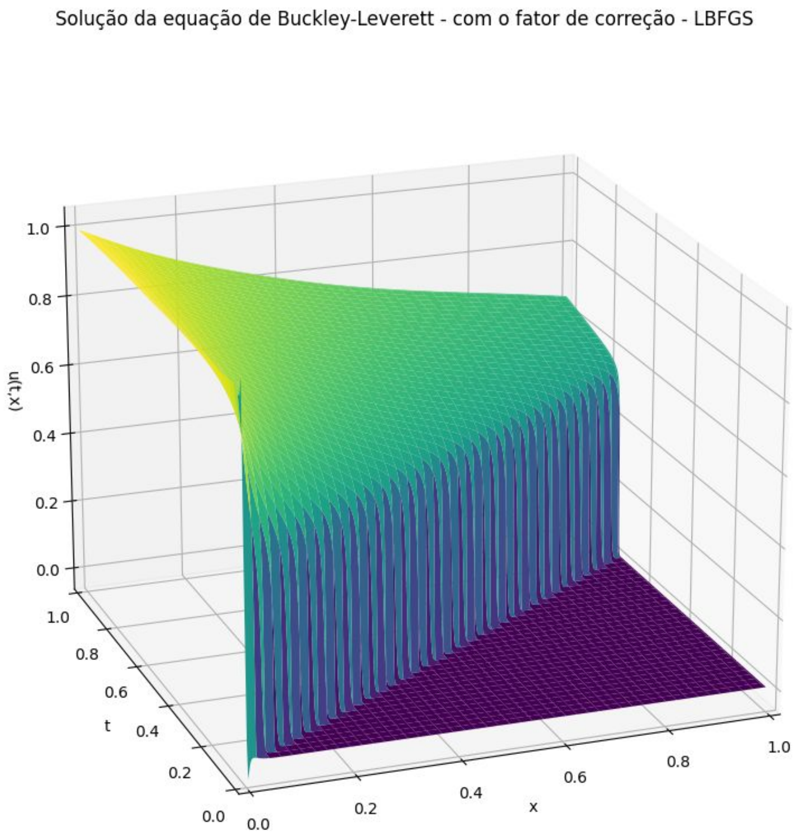
De acordo com Diab (2022), as PINNs são ineficazes em detectar descontinuidades bruscas nas EDPs, o que requer uma reformulação da equação para:

$$\frac{\partial S_w}{\partial t} + \frac{\partial f_w(S_w)}{\partial x} = \epsilon (S_w)_{xx},$$

Com **ε** sendo pequeno, do valor de 0,0025, assim a função gradiente do modelo é alterada para

Reformulação da EDP de Buckley-Leverett

```python
def gradiente(self):
  #Registrando tempo e posição para a diferenciação automática
    with tf.GradientTape(persistent=True) as tape:
      tape.watch(self.t)
      tape.watch(self.x)
      u = modelo(tf.stack([self.t[:,0], self.x[:,0]], axis=1))
      tape.watch(u)

      k_rw = ((u - Swi)/(1-Swi-Swo))**2
      k_ro = ((1- u - Swi)/(1-Swi-Swo))**2
      #f = (k_rw/u_w)/((k_rw/u_w) + (k_ro/u_o))
      f = fw(u, 2)
      f_u = tape.gradient(f,u)
      f_x = tape.gradient(f,self.x)

      ux = tape.gradient(u, self.x)
      ut = tape.gradient(u, self.t)
      uxx = tape.gradient(ux, self.x)

    del tape

    #return ut + ux*f_u
    return ut + f_x - (2.5e-3)*uxx
```



Caso de Buckley-Leverett
Otimizador L-BFGS-B