

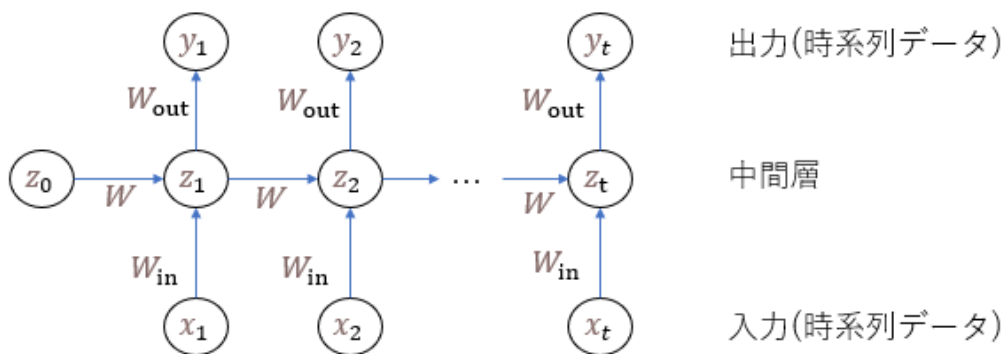
Section1: 再帰型ニューラルネットワークの概念

1. 要点まとめ

RNN(Recurrent Neural Network)とは、時系列データに対応可能なニューラルネットワークである。時系列データとは、時間的順序を追って一定間隔毎に観察され、相互に統計的依存関係が認められるデータ系列であり、例えば、音声データや自然言語などである。

RNNの構造は下図。時刻毎に入力 x_t 、中間層 z_t 、出力 y_t があり、中間層 z_t は1つ前の時刻の中間層 z_{t-1} とつながりがあることがRNNの大きな特徴である。これにより、時間的なつながりも特徴量として抽出できるようになる。重みは以下3つがあり、これが学習の調整対象となる。重みは全時刻共通である。

- 入力と中間層の間の重み W_{in}
- 中間層と出力の間の重み W_{out}
- 現在の中間層と1つ前の中間層の間の重み W



この構造を数式で表すと下式 (b,c: バイアス、f(), g(): 活性化関数)。

- $u_t = W_{in}x_t + Wz_{t-1} + b$
- $z_t = f(u_t)$
- $v_t = W_{out}z_t + c = W_{out}f(u_t) + c$
- $y_t = g(v_t)$

学習の際は、誤差逆伝播の派生版であるBPTT(Back Propagation Through Time)で重みを調整する。調整量を表す、誤差Eを重みやバイアスで微分した式は以下。

- $\frac{dE}{dW_{in}} = \sum_t \delta_t x_t \quad \times \delta_t = \frac{dE}{du_t}$
- $\frac{dE}{dW_{out}} = \sum_t \delta_{out,t} z_t \quad \times \delta_{out,t} = \frac{dE}{dv_t}$
- $\frac{dE}{dW} = \sum_t \delta_t z_{t-1}$
- $\frac{dE}{db} = \sum_t \delta_t$
- $\frac{dE}{dc} = \sum_t \delta_{out,t}$

上式で現れる $\delta_t, \delta_{out,t}$ (誤差Eを活性化関数前の中間出力 u_t, v_t で微分した値) の算出式は以下。

- $\delta_{out,t} = \frac{dE}{dy_t} g'(v_t)$
- $\delta_t = \delta_{out,t} \frac{dv_t}{du_t} = \delta_{out,t} W_{out} f'(u_t)$
- $\delta_{t-1} = \delta_t \frac{du_t}{du_{t-1}} = \delta_t W f'(u_{t-1})$

2. 実装演習

3_1_simple_RNN.ipynbのRNNのコードにおいて、下表No.1, No.2, No.3の重みの初期化方法&中間層の活性化関数の組み合わせを試行し、学習結果を元コードと比較する。

	重みの初期化	中間層の活性化関数
No.1	Xavier	シグモイド関数
No.2	Xavier	tanh関数
No.3	He	ReLU関数
元コード	標準正規分布	シグモイド関数

```
import numpy as np
from common import functions
import matplotlib.pyplot as plt

def d_tanh(x):
    return 1/(np.cosh(x) ** 2)

# データを用意
# 2進数の桁数
binary_dim = 8
# 最大値 + 1
largest_number = pow(2, binary_dim)
# largest_numberまで2進数を用意
binary = np.unpackbits(np.array([range(largest_number)], dtype=np.uint8).T, axis=1)

input_layer_size = 2
hidden_layer_size = 16
output_layer_size = 1

weight_init_std = 1
learning_rate = 0.1

iters_num = 10000
plot_interval = 100

# 試行パターン
trial_pattern = 3

# ウェイト初期化 (バイアスは簡単のため省略)
if trial_pattern == 0:
    trial_weight_initializer = "std"
    W_in = weight_init_std * np.random.randn(input_layer_size, hidden_layer_size)
    W_out = weight_init_std * np.random.randn(hidden_layer_size, output_layer_size)
    W = weight_init_std * np.random.randn(hidden_layer_size, hidden_layer_size)
elif trial_pattern == 1 or trial_pattern == 2:
    # Xavier
```

```

    trial_weight_initializer = "Xavier"
    W_in = np.random.randn(input_layer_size, hidden_layer_size) /
(np.sqrt(input_layer_size))
    W_out = np.random.randn(hidden_layer_size, output_layer_size) /
(np.sqrt(hidden_layer_size))
    W = np.random.randn(hidden_layer_size, hidden_layer_size) /
(np.sqrt(hidden_layer_size))
else:
    # He
    trial_weight_initializer = "He"
    W_in = np.random.randn(input_layer_size, hidden_layer_size) /
(np.sqrt(input_layer_size)) * np.sqrt(2)
    W_out = np.random.randn(hidden_layer_size, output_layer_size) /
(np.sqrt(hidden_layer_size)) * np.sqrt(2)
    W = np.random.randn(hidden_layer_size, hidden_layer_size) /
(np.sqrt(hidden_layer_size)) * np.sqrt(2)

# 活性化関数
func_activate = [functions.sigmoid, functions.sigmoid, np.tanh,
functions.relu]
func_d_activate = [functions.d_sigmoid, functions.d_sigmoid, d_tanh,
functions.d_relu]

if trial_pattern == 0 or trial_pattern == 1:
    trial_activation = "sigmoid"
elif trial_pattern == 2:
    trial_activation = "tanh"
else:
    trial_activation = "ReLU"

# 勾配
W_in_grad = np.zeros_like(W_in)
W_out_grad = np.zeros_like(W_out)
W_grad = np.zeros_like(W)

u = np.zeros((hidden_layer_size, binary_dim + 1))
z = np.zeros((hidden_layer_size, binary_dim + 1))
y = np.zeros((output_layer_size, binary_dim))

delta_out = np.zeros((output_layer_size, binary_dim))
delta = np.zeros((hidden_layer_size, binary_dim + 1))

all_losses = []

for i in range(iters_num):

    # A, B初期化 (a + b = d)
    a_int = np.random.randint(largest_number/2)
    a_bin = binary[a_int] # binary encoding
    b_int = np.random.randint(largest_number/2)
    b_bin = binary[b_int] # binary encoding

    # 正解データ
    d_int = a_int + b_int

```

```

d_bin = binary[d_int]

# 出力バイナリ
out_bin = np.zeros_like(d_bin)

# 時系列全体の誤差
all_loss = 0

# 順伝播
for t in range(binary_dim):
    # 入力値
    X = np.array([a_bin[-t-1], b_bin[-t-1]]).reshape(1, -1)

    # 時刻tにおける正解データ
    dd = np.array([d_bin[binary_dim - t - 1]])

    u[:,t+1] = np.dot(X, W_in) + np.dot(z[:,t].reshape(1, -1), W)
    z[:,t+1] = func_activate[trial_pattern](u[:,t+1])
    y[:,t] = functions.sigmoid(np.dot(z[:,t+1].reshape(1, -1), W_out))

    # 誤差
    loss = functions.mean_squared_error(dd, y[:,t])

    delta_out[:,t] = functions.d_mean_squared_error(dd, y[:,t]) *
functions.d_sigmoid(y[:,t])

    all_loss += loss

    out_bin[binary_dim - t - 1] = np.round(y[:,t])

# 逆伝播(BPTT)
for t in range(binary_dim)[::-1]:
    X = np.array([a_bin[-t-1], b_bin[-t-1]]).reshape(1, -1)

    delta[:,t] = (np.dot(delta[:,t+1].T, W.T) + np.dot(delta_out[:,t].T,
W_out.T)) * func_d_activate[trial_pattern](u[:,t+1])

    # 勾配更新
    W_out_grad += np.dot(z[:,t+1].reshape(-1,1), delta_out[:,t].reshape(-1,1))
    W_grad += np.dot(z[:,t].reshape(-1,1), delta[:,t].reshape(1,-1))
    W_in_grad += np.dot(X.T, delta[:,t].reshape(1,-1))

# 勾配適用
W_in -= learning_rate * W_in_grad
W_out -= learning_rate * W_out_grad
W -= learning_rate * W_grad

W_in_grad *= 0
W_out_grad *= 0
W_grad *= 0

if(i % plot_interval == 0):
    all_losses.append(all_loss)
    print("iters:" + str(i))

```

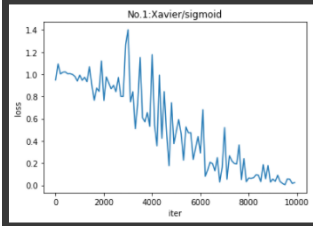
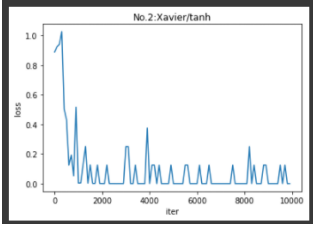
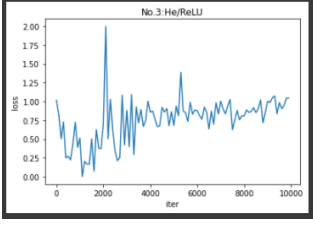
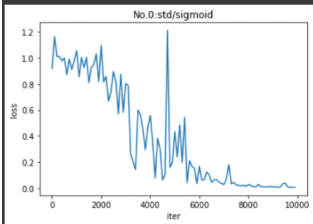
```
print("Loss:" + str(all_loss))
print("Pred:" + str(out_bin))
print("True:" + str(d_bin))

# out_binを10進数に直す→out_int
out_int = 0
for index,x in enumerate(reversed(out_bin)):
    out_int += x * pow(2, index)
print(str(a_int) + " + " + str(b_int) + " = " + str(out_int))
print("-----")

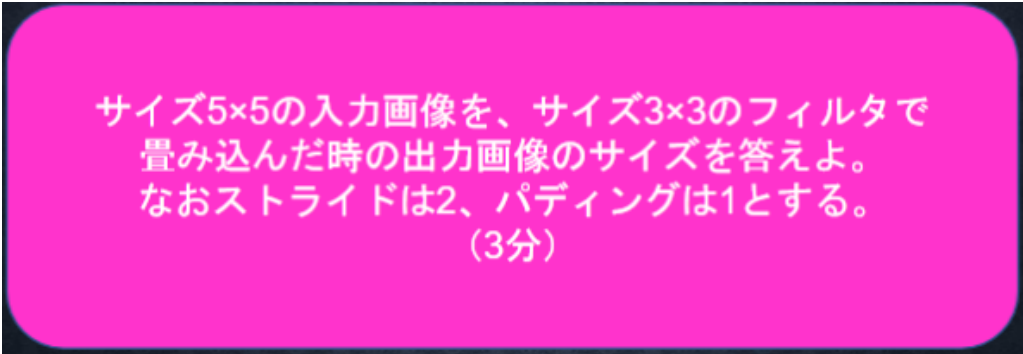
lists = range(0, iters_num, plot_interval)
plt.title("No."+str(trial_pattern)+":"+trial_weight_initializer + "/" +
trial_activation)
plt.xlabel("iter")
plt.ylabel("loss")
plt.plot(lists, all_losses, label="loss")
plt.show()
```

実行結果は以下

- 学習終了時に誤差(loss)が一番小さかったのはNo.2(Xavier & tanh)で、収束も早い
- No.1(Xavier & シグモイド)は、誤差が元コードと同等まで小さくなったものの、収束がやや遅め
- No.3(He & ReLU)は、
 - 誤差が減少しておらず、学習できていない。
 - 何回か動作させると、出力層のシグモイド関数がNaN(0割り)を返しエラー停止することがある。
 - 0割り(NaN)発生時は、勾配爆発が起きていると考えられる。

	重みの初 期化	中間層の活性化 関数	結果 (loss)	結果(グラフ)
No.1	Xavier	シグモイド関数	0.02378	<pre> ----- Iter:9900 Loss:0.023783768954660222 Pred:[0 1 1 0 0 1 1 1] True:[0 1 1 0 0 1 1 1] 57 + 46 = 103 ----- </pre> 
No.2	Xavier	tanh関数	0.00001	<pre> ----- Iter:9900 Loss:1.650211871098528e-05 Pred:[1 0 0 0 0 1 1] True:[1 0 0 0 0 1 1] 118 + 13 = 131 ----- </pre> 
No.3	He	ReLU関数	1.04456	<pre> ----- Iter:9900 Loss:1.0445691336933156 Pred:[0 0 0 0 0 0 0] True:[0 0 1 1 0 1 1] 49 + 0 = 0 ----- </pre> 
元コード	標準正規分布	シグモイド関数	0.00622	<pre> ----- Iter:9900 Loss:0.006220289168517406 Pred:[0 1 1 0 0 1 0] True:[0 1 1 0 0 1 0] 80 + 18 = 98 ----- </pre> 

3. 確認テスト



サイズ5×5の入力画像を、サイズ3×3のフィルタで
畳み込んだ時の出力画像のサイズを答えよ。
なおストライドは2、パディングは1とする。
(3分)

3x3

※動画では、2x2が正解、とあったが、これはフィルタサイズ5x5の場合の結果。フィルタサイズ3x3の場合は、出力画像3x3が正しい。

RNNのネットワークには大きくわけて3つの重みがある。1つは入力から現在の間層を定義する際にかけられる重み、1つは中間層から出力を定義する際にかけられる重みである。
残り1つの重みについて説明せよ。
(3分)

過去の間層から現在の間層を定義する際にかけられる重み。

演習チャレンジ

以下は再帰型ニューラルネットワークにおいて構文木を入力として再帰的に文全体の表現ベクトルを得るプログラムである。ただし、ニューラルネットワークの重みパラメータはグローバル変数として定義してあるものとし、`_activation`関数はなんらかの活性化関数であるとする。木構造は再帰的な辞書で定義しており、`root`が最も外側の辞書であると仮定する。

(`<`) にあてはまるのはどれか。

```
def traverse(node):
    """
    node: tree node, recursive dict, {left: node', right: node''}
    if leaf, word embeded vector, (embed_size,)
    """
    W: weights, global variable, (embed_size, 2*embed_size)
    b: bias, global variable, (embed_size,)
    """
    if not isinstance(node, dict):
        v = node
    else:
        left = traverse(node['left'])
        right = traverse(node['right'])
        v = _activation(
            (
            )
        )
    return v
```

- (1) `W.dot(left + right)`
- (2) `W.dot(np.concatenate([left, right]))`
- (3) `W.dot(left * right)`
- (4) `W.dot(np.maximum(left, right))`

(2) `W.dot(np.concatenate([left, right]))`

`traverse`は再帰的に文全体の表現ベクトルを得る関数のため、`left`, `right`の数値を保ったまま（四則演算等をしないで）結合する必要がある。よって、(2)が正解。

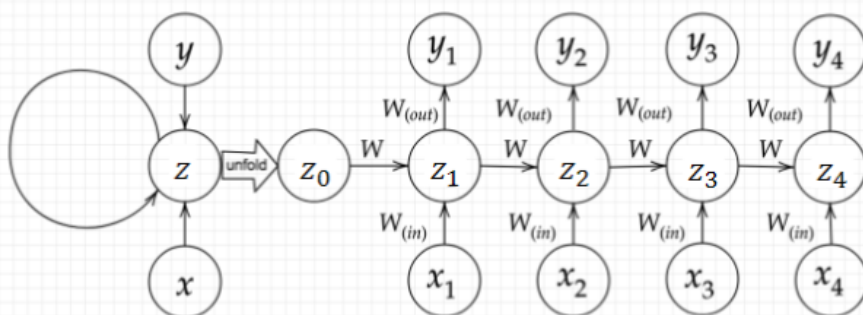
連鎖律の原理を使い、 dz/dx を求めよ。
(5分)

$$z = t^2$$

$$t = x + y$$

$$\frac{dz}{dx} = \frac{dz}{dt} \cdot \frac{dt}{dx} = 2t \cdot 1 = 2(x + y)$$

下図の y_1 を $x \cdot z_0 \cdot z_1 \cdot w_{in} \cdot w \cdot w_{out}$ を用いて数式で表せ。
 ※バイアスは任意の文字で定義せよ。
 ※また中間層の出力にシグモイド関数 $g(x)$ を作用させよ。
 (7分)



- $y_1 = g(W_{out}z_1 + c)$
- $z_1 = f(W_{in}x_1 + Wz_0 + b)$

コード演習問題

```
def bptt(xs, ys, W, U, V):
    """
    ys: labels, (batch_size, n_seq, output_size)
    """
    # 前向き
    hiddens, outputs = rnn_net(xs, W, U, V)
    # 損失関数のW, Y, Vに関する偏微分値
    dW = np.zeros_like(W) # dL/dW
    dU = np.zeros_like(U) # dL/dU
    dV = np.zeros_like(V) # dL/dV
    # 損失関数の出力値に関する偏微分値
    do = _calculate_do(outputs, ys) # dL/do, (batch_size, n_seq, output_size)

    batch_size, n_seq = ys.shape[:2]
    # 時間を逆方向にたどり、パラメータの偏微分値を計算 (バックプロパゲーション)
    # dL/dV = do/dV * dL/do = h * dL/do
    # dL/dW = do/dW * dL/do
    # dL/dU = do/dU * dL/do
    # do/dW = (dh_t/dW * d/dh_t + dh_{t-1}/dW * d/dh_{t-1} + ...) o_t
    # = (x_t + x_{t-1} * U + x_{t-2} * U^2 + ...) * V
    # do/dU = (dh_t/dU * d/dh_t + dh_{t-1}/dU * d/dh_{t-1} + ...) o_t
    # = (h_{t-1} + h_{t-2} * U + h_{t-3} * U^2 + ...) * V
    for t in reversed(range(n_seq)):
        dV += np.dot(do[:, t].T, hiddens[:, t]) / batch_size
        delta_t = do[:, t].dot(V)
        # 時間tの出力は時間t以前の中間層すべてに依存するため
        # W, Uはさらに遡って計算
        for bptt_step in reversed(range(t+1)):
            dW += np.dot(delta_t.T, xs[:, bptt_step]) / batch_size
            dU += np.dot(delta_t.T, hiddens[:, bptt_step-1]) / batch_size
            delta_t = (お)
    return dW, dU, dV
```

左の図はBPTTを行うプログラムである。なお簡単化のため活性化関数は恒等関数であるとする。

また、calculate_dout関数は損失関数を出力に関して偏微分した値を返す関数であるとする。

(お) にあてはまるのはどれか。

- (1) delta_t.dot(W)
- (2) delta_t.dot(U)
- (3) delta_t.dot(V)
- (4) delta_t * V

(お) は、以下漸化式を実装した箇所となる。

$$\bullet \delta_{t-1} = \delta_t \frac{du_t}{du_{t-1}}$$

$\frac{du_t}{du_{t-1}} = U$ であるため、正解は「(2) delta_t.dot(U)」である。