深層学習day3-4　レポート

深層学習day3　確認テスト

1.
3×3

2
前の中間層から現在の中間層を定義する際にかけられる重み

3
2(x+y)

4
y1=g(Wout*z1+bout)
z1=f(W*z0+Win*x1+bin)

5
（２）

6
忘却ゲート

7
LSTM　パラメーターが多い
CEC　勾配を持たない

8
パラメーターの数が少ないためGRUの方が速い

9
(2)

10
seq2seq　１文に対して１文を返す
HRED　文脈も判断材料に加えて１文を返す
VHRED　ありきたりな回答に収束しない工夫がされている。

11
確率変数

12
RNN　時系列データの学習手法
word2vec　文章から、単語の意味ベクトルを得る
Seq2seq　時系列データから時系列データを返す手法
Attention　時系列データの中の重要な内部特徴量を取り出す重みを学習する手法

深層学習day3　要点まとめ

・Section 1　再帰型ニューラルネットワーク
　時系列データを扱うNNである。時系列のデータを中間層に蓄積していきながら、未来の時系列を予測する。学習にはBPTTというアルゴリズムを用いるが、NNの場合の誤差逆伝播と一緒で、基本的には後ろ向きに偏微分をしていくが、出力層の重み以外は順伝播の途中で何度も出てくるため、それらについて相加的な処理を行う。

・Section 2　LSTM
　CECという記憶する層を持っておくことによって、より昔の時系列についての学習効率をよくしている。一方でこの記憶する層では時刻歴に対して重みが変わらないため、何を覚えさせるか、何を忘れさせるかを学習するノードによって機能を補っている。

・Section 3　GRU
　LSTMはRNNの課題を解決している一方で、パラメーター数が多いので学習が重いという課題がある。GRUではその原因であるCECを排して、入力ゲートと出力ゲートを代替するようゲート（リセットゲート、更新ゲート）によって機能をなしている。

・Section 4　双方向RNN
　通常のRNNに対して、未来の情報を加味することによって学習精度をあげているモデル。特に文章系の学習に多く利用される傾向にある。これは入力時に過去から未来のデータを全て入力する必要があるため、応用範囲は限られるが、予測精度はあがる。

・Section 5　Seq2seq
　時系列データから時系列データを返す手法である。エンコーダー部分とデコーダー部分にわかれており、入力データを解釈する部分と、解釈から出力データを生成する部分にタスクを分けている。ここでいう解釈というのは、入力の文のデータに対し、分割されたトークンに対し分散表現ベクトルを割り当てている。
　このアルゴリズムだけだと文脈判断が出来ないので、HREDなどのアルゴリズムも存在する。

・Section 6　Word2vec
　RNNに単語のような可変長の文字列を食わせるために、固定長形式に書き換える方法である。ボキャブラリに対して任意の単語との相関量のような形で重み行列を作る。これによって計算負荷を従来より下げることができた。

・Section 7　Attention Mechanism
　入力と出力の単語の関連度に関して学習をする仕組みである。具体的には候補となっている単語のベクトル表現に対して、どこに注視するかという重みづけを確率で評価し、情報を持つ変数があり、内積をとることで単語を選択している。

深層学習day4　要点まとめ

・Section 1　強化学習

　強化学習とは、報酬を最大化する方策を環境から学習して、最適な行動を決定していくモデルである。機械学習としては、教師あり学習や教師なし学習とは違う種類の学習であり、値の推測というよりは、利益を最大化する戦略を学習することに特化している。

・Section 2　AlphaGo

　AlphaGoは囲碁に関して強化学習を施したものである。方策関数と価値関数に深層学習を用いている。また初期のモデルでは学習を早めるために、入力データには囲碁の戦略について既知なアイデアも入れており、また、方策の学習には棋譜データの教師あり学習であらかじめ学習したものを転移させた。一方新しいモデルではそのようなヒューリスティックな要素を排している。また、方策関数と価値観数を結合しているため、層が厚くなっており、Residual Network という、層をショートカットして学習を行う工夫もされている。

・Section 3　軽量化・高速化技術

　深層学習では特に、パラメーター更新の数が多かったり、教師データが多かったりと計算量が多くなりがちなので、それらに対する解決方法が必要である。大きく分けると方法は２つあり、一つはCPUやGPUというようなハードそのものを改良する方法であり、もう一つは、並列化する方法である。並列化するにはアルゴリズムそのものの改良が必要であり、モデルそのものを分けたり、教師データを分割して更新する方法を工夫したりすることがある。

　また、IoTのように容量が小さいデバイスで推論をしたり、速度を上げたい場合は、モデルそのものを軽くする必要がある。この場合精度と速度がトレードオフになることに注意が必要である。量子化という方法では、浮動小数点の一部をカットしたり、蒸留という方法では、深い層の推論を教師データとして小さいモデルに学習させたりする。

・Section 4　応用モデル

　軽量化・高速化という課題に対して、特にCNNのコンボリューション層の学習を効率化する試みが、MobileNetという方法では試みられている。これはコンボリューション層をレイヤー枚に対する平面的な特徴抽出と、レイヤをまたぐ間での特徴抽出の２つの意味合いがあると解釈し、それらを切り離すことによって計算量を減らそうという試みである。

　また、DenseNetという方法では、層が厚くなる学習について勾配消失塔を防ぐ方法の一つで、部分的に前の層の出力を新しい入力にどんどん重ね合わせていくフェーズを織り込むことで情報を落とさない工夫がされている。

・Section 5　Transformer

　エンコーダー・デコーダーモデルの一つで、翻訳に用いられる。入力の言語に足しして意味解釈をエンコーダーで行い、得られた意味解釈にたいして、対応する多言語の単語を組み合わせて文として返す仕組みをデコーダーで行っている。この時、ある単語に対して関連の深い単語を、Attentionメカニズムを用いて推測し、文章を構成している。

・Section 6　物体検知・セグメンテーション

　object　detectionと一口に言ってもその種類は多様であり、画像一枚に対してラベルを降る'分類'、画像の中から物体をBounding boxで抜き出して検出する'物体検知'、ピクセルに対してラベルを振る'意味領域分割'、さらにそこからそれぞれの個体領域まで判別する'個体領域分割'、というように、得られる出力も難易度も異なる。代表的なデータセットがいくつかあり、それらによってモデルが評価されることが多いが、データセットごとに特徴があり、その趣旨はしっかり見極める必要がある。モデルの評価方法は複雑だが、IoUという独特な概念があり（物体の位置の正確さを表す）、それと、クラスの整合度を絡めた評価を行う。

RNNの実装

In [ ]:

In [2]:
```
import numpy as np
from common import functions
import matplotlib.pyplot as plt
```

In [3]:
```
def d_tanh(x):
    return 1/(np.cosh(x) ** 2)
```

In [16]:
```
# 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 = 32
output_layer_size = 1

weight_init_std = 1
learning_rate = 0.1

iters_num = 10000
plot_interval = 100

# ウェイト初期化 (バイアスは簡単のため省略)
# 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)
# Xavier
# W_in = np.random.randn(input_layer_size, hidden_layer_size) / (np.sqrt(input_layer_s
# W_out = np.random.randn(hidden_layer_size, output_layer_size) / (np.sqrt(hidden_laye
# W = np.random.randn(hidden_layer_size, hidden_layer_size) / (np.sqrt(hidden_layer_si
# He
W_in = np.random.randn(input_layer_size, hidden_layer_size) / (np.sqrt(input_layer_si
W_out = np.random.randn(hidden_layer_size, output_layer_size) / (np.sqrt(hidden_layer
W = np.random.randn(hidden_layer_size, hidden_layer_size) / (np.sqrt(hidden_layer_siz


# 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

    # 時系列ループ
    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] = functions.sigmoid(u[:,t+1])
        z[:,t+1] = functions.relu(u[:,t+1])
#         z[:,t+1] = np.tanh(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_sig

        all_loss += loss

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


    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
        delta[:,t] = (np.dot(delta[:,t+1].T, W.T) + np.dot(delta_out[:,t].T, W_out.T)
#         delta[:,t] = (np.dot(delta[:,t+1].T, W.T) + np.dot(delta_out[:,t].T, W_out.T

        # 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))
```

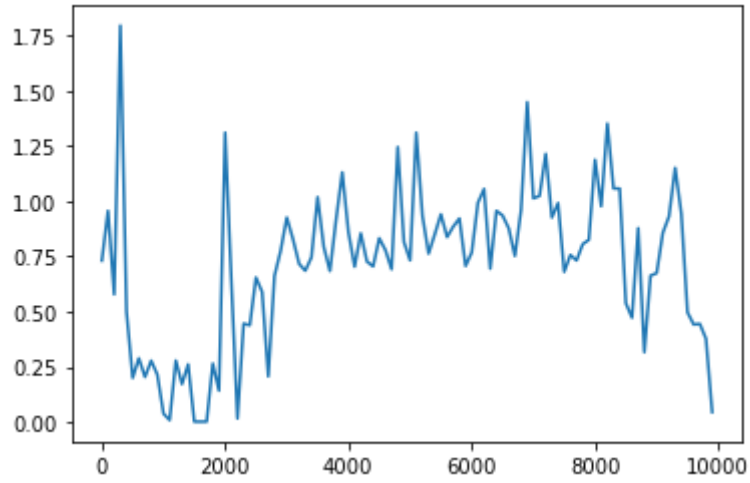
```
#         print("Loss:" + str(all_loss))
#         print("Pred:" + str(out_bin))
#         print("True:" + str(d_bin))
        out_int = 0
        for index,x in enumerate(reversed(out_bin)):
            out_int += x * pow(2, index)
        print(str(a_int) + " + " + str(b_int) + " = " + str(out_int))
#         print("------------")
```

```
93 + 35 = 76
29 + 61 = 2
98 + 68 = 38
107 + 27 = 254
120 + 87 = 15
8 + 114 = 122
33 + 105 = 202
98 + 25 = 123
126 + 127 = 253
104 + 111 = 215
20 + 39 = 59
48 + 13 = 61
124 + 24 = 148
42 + 20 = 62
58 + 14 = 72
17 + 29 = 46
30 + 33 = 63
83 + 90 = 173
108 + 88 = 196
98 + 46 = 144
127 + 73 = 146
55 + 53 = 238
19 + 86 = 105
123 + 121 = 118
43 + 85 = 130
13 + 29 = 174
3 + 65 = 70
14 + 25 = 35
38 + 6 = 172
22 + 93 = 107
80 + 119 = 5
74 + 29 = 85
120 + 42 = 130
25 + 34 = 59
4 + 33 = 37
24 + 107 = 33
5 + 104 = 37
120 + 3 = 41
100 + 49 = 85
41 + 92 = 85
34 + 52 = 18
37 + 14 = 33
44 + 67 = 37
53 + 18 = 5
15 + 36 = 33
120 + 34 = 10
79 + 100 = 33
73 + 105 = 32
82 + 63 = 5
58 + 98 = 0
109 + 106 = 5
70 + 58 = 40
92 + 70 = 10
36 + 47 = 1
34 + 2 = 32
7 + 53 = 16
34 + 69 = 37
34 + 117 = 85
57 + 83 = 40
64 + 90 = 10
52 + 106 = 10
108 + 14 = 34
44 + 56 = 20
102 + 9 = 37
94 + 92 = 2
88 + 123 = 33
34 + 125 = 85
16 + 5 = 21
57 + 105 = 0
```

```
123 + 54 = 69
122 + 90 = 32
69 + 47 = 32
121 + 28 = 65
90 + 108 = 34
56 + 42 = 2
41 + 78 = 37
105 + 48 = 9
83 + 90 = 9
30 + 33 = 21
13 + 92 = 81
23 + 57 = 4
19 + 122 = 41
55 + 122 = 69
34 + 63 = 21
107 + 103 = 28
86 + 5 = 83
4 + 66 = 70
86 + 73 = 7
21 + 83 = 72
35 + 42 = 9
9 + 71 = 192
47 + 39 = 144
126 + 114 = 0
66 + 47 = 237
15 + 27 = 32
108 + 59 = 131
87 + 39 = 92
68 + 88 = 148
79 + 122 = 203
117 + 11 = 128
```

In [17]:

```
lists = range(0, iters_num, plot_interval)
plt.plot(lists, all_losses, label="loss")
plt.show()
```



RNNの実装を体験した。中間層や重みの更新を変えてもあまり大きな影響はなかったが、学習率によってはうまく収束せず、ReLUを使っているときはその傾向が顕著であることを確認できた。