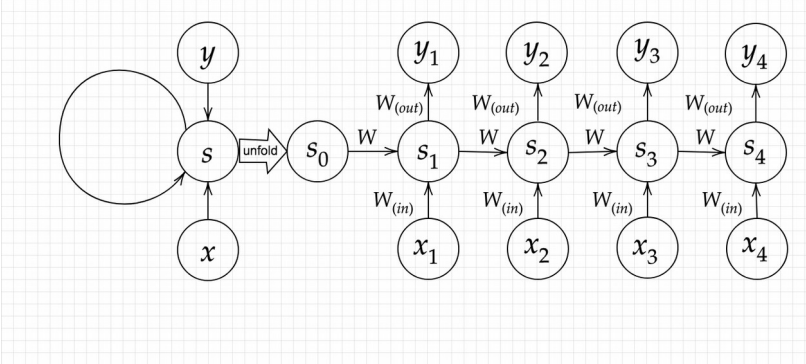
深層学習（後編１）

Ｓｅｃｔｉｏｎ１）再帰型ニューラルネットワークの概念

１－１　ＲＮＮ全体像
１－１－１　ＲＮＮ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
１－１－２　時系列データ
・時間的順序を追って一定間隔ごとに観察され、相互に統計的依存関係が認められるデータの系列
・具体例：音声データ、テキストデータ、株価データ
１－１－３　ＲＮＮについて
・中間層が重要



・ＲＮＮの数学的記述
$u^t = W_{(in)} x^t + W z^{t-1} + b$
$z^t = f(W_{(in)} x^t + W z^{t-1} + b)$
$v^t = W_{(out)} z^t + c$
$y^t = g(W_{(out)} z^t + c)$
・ＲＮＮの３つの重み
－入力から現在の中間層を定義する際にかけられる重み
－中間層から出力層を定義する際にかけられる重み
－中間層から次の中間層へ渡される際にかけられる重み（上の「W」）
・ＲＮＮの特徴
初期の状態と過去の時間ｔ－１の状態を保持し、そこから次の時間ｔでの状態を再帰的に求める再帰構造が必要となること
・深層学習３－３～実装演習

In [2]:

```
import sys, os
sys.path.append(os.pardir)  #親ディレクトリのファイルをインポートするための設定
import numpy as np
from common import functions
import matplotlib.pyplot as plt

# def d_tanh(x):



#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
#hidden_layer_size = 32
output_layer_size = 1

weight_init_std = 1
learning_rate = 0.1
#weight_init_std = 2
#learning_rate = 0.05

```

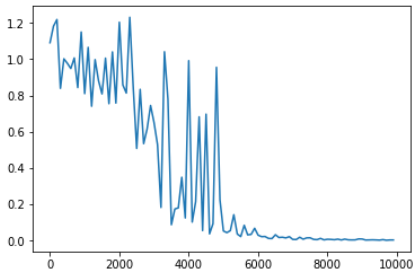
```
iters_num = 10000
plot_interval = 100

# 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


# He



# 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])

        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])


    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)) * functions.d_sigmoid(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


```

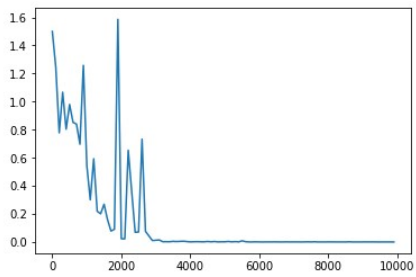
```
    if(i % plot_interval == 0):
        all_losses.append(all_loss)
        print("iters:" + str(i))
        print("Loss:" + str(all_loss))
        print("Pred:" + str(out_bin))
        print("True:" + str(d_bin))
        out_int = 0
        for index,x in enumerate(reversed(out_bin)):
            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.plot(lists, all_losses, label="loss")
plt.show()
```

```
Pred:[1 0 0 0 0 1 0 1]
True:[1 0 0 0 0 1 0 1]
43 + 90 = 133
------------
```



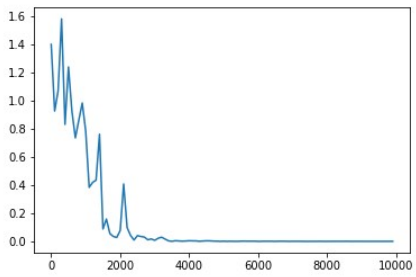
－ｈｉｄｄｅｎ＿ｌａｙｅｒ＿ｓｉｚｅ、ｗｅｉｇｈｔ＿ｉｎｉｔ＿ｓｔｄ、ｌｅａｒｎｉｎｇ＿ｒａｔｅの変更

```
input_layer_size = 2
#hidden_layer_size = 16
hidden_layer_size = 32
output_layer_size = 1

#weight_init_std = 1
#learning_rate = 0.1
weight_init_std = 2
learning_rate = 0.2
```



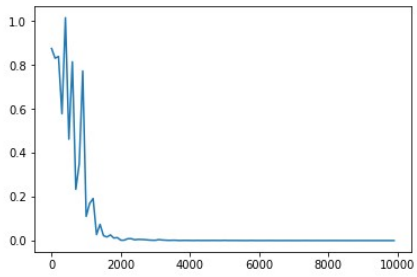
－重みの初期化方法の変更（Ｘａｖｉｅｒ）

```
# 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 =  weight_init_std * np.random.randn(input_layer_size, hidden_layer_size) / (np.sqrt(input_layer_size))
W_out = weight_init_std * np.random.randn(hidden_layer_size, output_layer_size)  / (np.sqrt(hidden_layer_size))
W = weight_init_std * np.random.randn(hidden_layer_size, hidden_layer_size)  / (np.sqrt(hidden_layer_size))

```



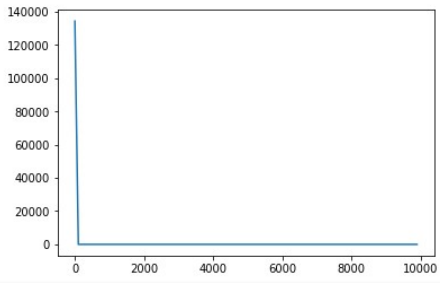
－重みの初期化方法の変更（Ｈｅ）

```
#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 =  weight_init_std * np.random.randn(input_layer_size, hidden_layer_size) / (np.sqrt(input_layer_size))
#W_out = weight_init_std * np.random.randn(hidden_layer_size, output_layer_size)  / (np.sqrt(hidden_layer_size))
#W = weight_init_std * np.random.randn(hidden_layer_size, hidden_layer_size)  / (np.sqrt(hidden_layer_size))

# He
W_in =  weight_init_std * np.random.randn(input_layer_size, hidden_layer_size) / (np.sqrt(input_layer_size)) * np.sqrt(2)
W_out = weight_init_std * np.random.randn(hidden_layer_size, output_layer_size)  / (np.sqrt(hidden_layer_size)) * np.sqrt(2)
W = weight_init_std * np.random.randn(hidden_layer_size, hidden_layer_size)  / (np.sqrt(hidden_layer_size)) * np.sqrt(2)

```



・深層学習３－４～実装演習
－活性化関数のＲｅＬＵへの変更

```
        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])

        #y[:,t] = functions.sigmoid(np.dot(z[:,t+1].reshape(1, -1), W_out))
        y[:,t] = functions.relu(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])
        delta_out[:,t] = functions.d_mean_squared_error(dd, y[:,t]) * functions.d_relu(y[:,t])

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

```



ー活性化関数のｔａｎｈへの変更

jupyter functions.py✔ 7分前

Logout

File Edit View Language Python

```python
import numpy as np

# 中間層の活性化関数
# シグモイド関数（ロジスティック関数）
def sigmoid(x):
    return 1/(1 + np.exp(-x))

# ReLU関数
def relu(x):
    return np.maximum(0, x)

# ステップ関数（閾値0）
def step_function(x):
    return np.where( x > 0, 1, 0)

#tarh関数
def tanh(x):
    return np.tanh(x)


# ReLU関数の導関数
def d_relu(x):
    return np.where( x > 0, 1, 0)

# tanh関数の導関数
def d_tanh(x):
    dx = 1 / (np.cosh(x)**2)
    return dx

# ステップ関数の導関数
def d_step_function(x):
    return 0
```

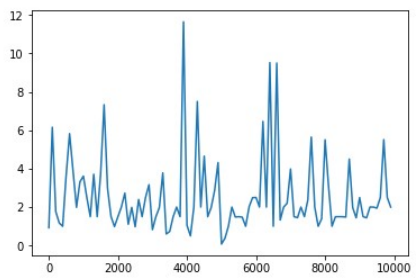
jupyter 3_1_simple_RNN Last Checkpoint: 2019/06/08 (autosaved)

Logout

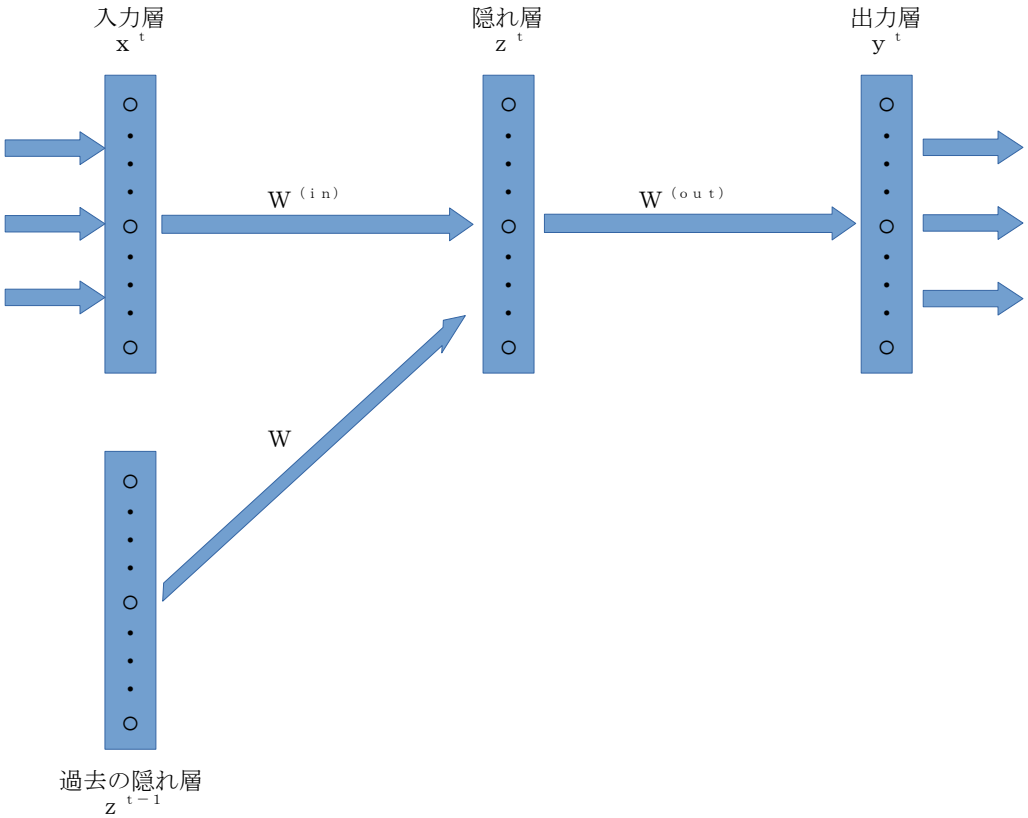
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Markdown

```python
        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] = functions.tanh(u[:,t+1])

        #y[:,t] = functions.sigmoid(np.dot(z[:,t+1].reshape(1, -1), W_out))
        #y[:,t] = functions.relu(np.dot(z[:,t+1].reshape(1, -1), W_out))
        y[:,t] = functions.tanh(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])
        #delta_out[:,t] = functions.d_mean_squared_error(dd, y[:,t]) * functions.d_relu(y[:,t])
        delta_out[:,t] = functions.d_mean_squared_error(dd, y[:,t]) * functions.d_tanh(y[:,t])

        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)) * functions.d_sigmoid(u[:,t+1])
        #delta[:,t] = (np.dot(delta[:,t+1].T, W.T) + np.dot(delta_out[:,t].T, W_out.T)) * functions.d_relu(u[:,t+1])
        delta[:,t] = (np.dot(delta[:,t+1].T, W.T) + np.dot(delta_out[:,t].T, W_out.T)) * functions.d_tanh(u[:,t+1])
```



１－２　ＢＰＴＴ
１－２－１　ＢＰＴＴとは
・誤差逆伝播の一種
・Ｂａｃｋ　Ｐｒｏｐａｇａｔｉｏｎ　Ｔｈｒｏｕｇｈ　Ｔｉｍｅ
１－２－２　ＢＰＴＴの数学的記述



・モデルの出力を表す式。W (in)、W (out) があることが特徴

$$u^t = W_{(in)}x^t + W\,z^{t-1} + b$$
$$z^t = f\left(W_{(in)}x^t + W\,z^{t-1} + b\right)$$
$$v^t = W_{(out)}z^t + c$$
$$y^t = g\left(W_{(out)}z^t + c\right)$$

・誤差関数のそれぞれのパラメータ（重み、バイアス）に対する勾配
δ は隠れ層、出力層における誤差項を定義

$$\frac{\partial E}{\partial W_{(in)}} = \frac{\partial E}{\partial u^t}\left[\frac{\partial u^t}{\partial W_{(in)}}\right]^T = \delta^t\left[x^t\right]^T$$
$$\frac{\partial E}{\partial W_{(out)}} = \frac{\partial E}{\partial v^t}\left[\frac{\partial v^t}{\partial W_{(out)}}\right]^T = \delta^{out,t}\left[z^t\right]^T$$
$$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial u^t}\left[\frac{\partial u^t}{\partial W}\right]^T = \delta^t\left[z^{t-1}\right]^T$$
$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial u^t}\frac{\partial u^t}{\partial b} = \delta^t$$
$$\frac{\partial E}{\partial c} = \frac{\partial E}{\partial v^t}\frac{\partial v^t}{\partial c} = \delta^{out,t}$$

・時刻ｔ－１における誤差を求める。δ^{t-1}をδ^{t}の式で表す

$$\delta^{t-1} = \frac{\partial E}{\partial u^{t-1}} = \frac{\partial E}{\partial u^{t}}\frac{\partial u^{t}}{\partial u^{t-1}} = \delta^{t}\left\{\frac{\partial u^{t}}{\partial z^{t-1}}\frac{\partial z^{t-1}}{\partial u^{t-1}}\right\} = \delta^{t}\left\{Wf'\left(u^{t-1}\right)\right\}$$

$$\delta^{t-z-1} = \delta^{t-z}\left\{Wf'\left(u^{t-z-1}\right)\right\}$$

・各パラメータの更新式

$$W_{(in)}^{t+1} = W_{(in)}^{t} - \epsilon\frac{\partial E}{\partial W_{(in)}} = W_{(in)}^{t} - \epsilon\sum_{z=0}^{T_t}\delta^{t-z}\left[\, x^{t-z}\,\right]^{T}$$

$$W_{(out)}^{t+1} = W_{(out)}^{t} - \epsilon\frac{\partial E}{\partial W_{(out)}} = W_{(in)}^{t} - \epsilon\delta^{out,t}\left[\, z^{t}\,\right]^{T}$$

$$W^{t+1} = W^{t} - \epsilon\frac{\partial E}{\partial W} = W_{(in)}^{t} - \epsilon\sum_{z=0}^{T_t}\delta^{t-z}\left[\, z^{t-z-1}\,\right]^{T}$$

$$b^{t+1} = b^{t} - \epsilon\frac{\partial E}{\partial b} = b^{t} - \epsilon\sum_{z=0}^{T_t}\delta^{t-z}$$

$$c^{t+1} = c^{t} - \epsilon\frac{\partial E}{\partial c} = c^{t} - \epsilon\delta^{out,t}$$

１－２－３　ＢＰＴＴの全体像
・δ^{t-1}とδ^{t}の関係に注意すること（上記数式参照）

Ｓｅｃｔｉｏｎ２）ＬＳＴＭ
・ＲＮＮの課題
　時系列を遡れば遡るほど勾配が消失
　→ＲＮＮの構造を変えて解決したものがＬＳＴＭ
２－１　ＣＥＣ
・勾配消失・勾配爆発・・・勾配が１であり続ければ解決できる
２－２　入力ゲートと出力ゲート
・ＣＥＣの課題：入力データについて時間依存度に関係なく重みが一律である
　→同じ重みでつながっているため重みの更新時にお互いを打ち消し合ってしまう
　　（入力重み衝突、出力重み衝突）
　→入力ゲートと出力ゲートを追加することでそれぞれのゲートへの入力値の重みを可変可能とする
２－３　忘却ゲート
・ＣＥＣの課題：ＣＥＣに保管された情報が不要となった場合も削除できない
　→忘却ゲートを追加することで過去の情報が不要となった場合、そのタイミングで情報を忘却する
２－４　覗き穴結合
・ＣＥＣの課題：ゲートの制御にＣＥＣ自身が保持している値が用いられていない
　→ＣＥＣと各ゲートをつなぎ、ＣＥＣの状態を各ゲートに伝える

Ｓｅｃｔｉｏｎ３）ＧＲＵ
・ＬＳＴＭの課題
　パラメータが多く、計算負荷が高い
　→ＧＲＵはパラメータを大幅に削減し、精度は同等またはそれ以上が望める
・ＧＲＵの構造
　ＬＳＴＭはＣＥＣ、入力ゲート、出力ゲート、忘却ゲートで構成
　⇔ＧＲＵはリセットゲート、更新ゲートのみで構成
※ＬＳＴＭとＧＲＵどちらがいいか・・・やってみること

Ｓｅｃｔｉｏｎ４）双方向ＲＮＮ
・過去の情報だけではなく未来の情報も加味することで精度の向上を目指すモデル
・文章推敲、機械翻訳に応用

Ｓｅｃｔｉｏｎ５）Ｓｅｑ２Ｓｅｑ
・Ｅｎｃｏｄｅｒ－Ｄｅｃｏｄｅｒモデルの一種
・機械対話、機械翻訳に応用

５－１　Ｅｎｃｏｄｅｒ　ＲＮＮ
・Ｅｎｃｏｄｅｒ　ＲＮＮ
文章を単語などのトークン毎に分割しＩＤを付与（Ｔａｋｉｎｇ）
ＩＤからそのトークンを表す分散表現ベクトルに変換（Ｅｍｂｅｄｄｉｎｇ）
ベクトルを順番にＲＮＮに入力（Ｅｎｃｏｄｅｒ　ＲＮＮ）
・処理手順
－ｖｅｃ１をＲＮＮに入力し、ｈｉｄｄｅｎ　ｓｔａｔｅを出力
このｈｉｄｄｅｎ　ｓｔａｔｅと次の入力ｖｅｃ２をまたＲＮＮに入力し
ｈｉｄｄｅｎ　ｓｔａｔｅを出力
上記を繰り返す
－最後のｖｅｃを入れたときのｈｉｄｄｅｎ　ｓｔａｔｅをｆｉｎａｌ　ｓｔａｔｅ
（ｔｈｏｕｇｈｔ　ｖｅｃｔｏｒ／入力した文の意味を表すベクトル）とする
５－２　Ｄｅｃｏｄｅｒ　ＲＮＮ
・Ｄｅｃｏｄｅｒ　ＲＮＮ
システムがアウトプットデータを単語等のトークン毎に生成
・Ｄｅｃｏｄｅｒ　ＲＮＮ
①ｆｉｎａｌ　ｓｔａｔｅ（ｔｈｏｕｇｈｔ　ｖｅｃｔｏｒ）から各ｔｏｋｅｎの生成確率を出力
ｆｉｎａｌ　ｓｔａｔｅをＤｅｃｏｄｅｒ　ＲＮＮのｉｎｉｔｉａｌ　ｓｔａｔｅとして設定し
Ｅｍｂｅｄｄｉｎｇを入力（Ｄｅｃｏｄｅｒ　ＲＮＮ）
②生成確率に基づいてｔｏｋｅｎをランダムに選択（Ｓａｍｐｌｉｎｇ）
③②で選択されたｔｏｋｅｎをＥｍｂｅｄｄｉｎｇしＤｅｃｏｄｅｒ　ＲＮＮの次の入力とする
④①～③を繰り返し②で得られたｔｏｋｅｎを文字列に直す（Ｄｅｔｏｋｅｎｉｚｅ）
５－３　ＨＲＥＤ
・Ｓｅｑ２Ｓｅｑの課題
１問１答しかできない
会話の文脈無視で応答される
・ＨＲＥＤ
過去の発話から次の発話を生成
前の単語の流れに即して応答されるためより人間らしい文章が生成される
・ＨＲＥＤの構造
ＨＲＥＤ＝Ｓｅｑ２Ｓｅｑ＋Ｃｏｎｔｅｘｔ　ＲＮＮ
Ｃｏｎｔｅｘｔ　ＲＮＮ：Ｅｎｃｏｄｅｒのまとめた各文章系列をこれまでの会話コンテキスト
全体を表すベクトルに変換
→過去の発話履歴を加味した返答となる
・ＨＲＥＤの課題
－会話の「流れ」のような多様性がない
－短く情報量に乏しい回答をしがち
５－４　ＶＨＲＥＤ
・ＨＲＥＤの課題を解決するため、ＨＲＥＤにＶＡＥの潜在変数の概念を追加したもの
５－５　ＶＡＥ
５－５－１　オートエンコーダー
・オートエンコーダーとは
教師なし学習のひとつ
Ｅｎｃｏｄｅｒ：入力データから潜在変数ｚに変換するニューラルネットワーク
Ｄｅｃｏｄｅｒ：潜在変数ｚから元画像を復元するニューラルネットワーク
メリットは次元削減を行えること
５－５－２　ＶＡＥ
通常のオートエンコーダーではデータを潜在変数ｚに押し込めている
→その構造がどういう状態かわからない
→ＶＡＥは潜在変数ｚに確率分布ｚ～Ｎ（０、１）を仮定したもの

Ｓｅｃｔｉｏｎ６）Ｗｏｒｄ２ｖｅｃ
・ＲＮＮの課題
単語のような可変長の文字列をニューラルネットワークに与えることができない
固定長で単語を表す必要
・ｗｏｒｄ２ｖｅｃ
学習データからボキャブラリを作成
単語はｏｎｅ－ｈｏｔベクトルで表される
・ｗｏｒｄ２ｖｅｃのメリット
重み行列がボキャブラリ数×任意の単語ベクトル次元で表される
（ＲＮＮではボキャブラリ数×ボキャブラリ数で表される）
→大規模データの分散表現の学習が現実的な計算速度とメモリ量で実現可能に

Ｓｅｃｔｉｏｎ７）ＡｔｔｅｎｔｉｏｎＭｅｃｈａｎｉｓｍ
・Ｓｅｑ２Ｓｅｑの課題
　長い文章への対応が難しい
　単語数が多くなっても固定次元ベクトルの中に入力しなければならない
　→文章が長くなるほどそのシーケンスの内部表現の次元も大きくなっていく仕組みが必要
・ＡｔｔｅｎｔｉｏｎＭｅｃｈａｎｉｓｍ
　「入力と出力のどの単語が関連しているのか」を学習する仕組み

深層学習（後編２）

・深層学習４－２～実装演習
　定数を定義（tf.constant）
　セッションを定義（tf.Session）してラン（sess.run）することでテンソルに値が入る
　セッションの定義前に定数を表示すると実際の値ではなくテンソルの形状が表示される

constant

In [1]:
```
import tensorflow as tf
import numpy as np

# それぞれ定数を定義
a = tf.constant(1)
b = tf.constant(2, dtype=tf.float32, shape=[3,2])
c = tf.constant(np.arange(4), dtype=tf.float32, shape=[2,2])

print('a:', a)
print('b:', b)
print('c:', c)

sess = tf.Session()

print('a:', sess.run(a))
print('b:', sess.run(b))
print('c:', sess.run(c))
```

```
a: Tensor("Const:0", shape=(), dtype=int32)
b: Tensor("Const_1:0", shape=(3, 2), dtype=float32)
c: Tensor("Const_2:0", shape=(2, 2), dtype=float32)
a: 1
b: [[2. 2.]
 [2. 2.]
 [2. 2.]]
c: [[0. 1.]
 [2. 3.]]
```

　プレースホルダを定義（tf.placeholder）すると後から値を変更できる
　バッチの際に使用される（ｘをバッチ毎に代入する時など）

placeholder

In [2]:
```
import tensorflow as tf
import numpy as np

#プレースホルダーを定義
x = tf.placeholder(dtype=tf.float32, shape=[None,3])

print('x:', x)

sess = tf.Session()

X = np.random.rand(2,3)
print('X:', X)

#プレースホルダにX[0]を入力
#shapeを(3,)から(1,3)にするためreshape
print('x:', sess.run(x, feed_dict={x:X[0].reshape(1,-1)}))
#プレースホルダにX[1]を入力
print('x:', sess.run(x, feed_dict={x:X[1].reshape(1,-1)}))
```

```
x: Tensor("Placeholder:0", shape=(?, 3), dtype=float32)
X: [[0.16016602 0.42413725 0.7493061 ]
 [0.58616345 0.72662008 0.73221939]]
x: [[0.16016603 0.42413723 0.7493061 ]]
x: [[0.58616346 0.7266201  0.7322194 ]]
```

変数を定義（tf.Variable）
変数を初期化（tf.global_variables_initializer）しラン（sess.run）することで変数に値が入る

variables

In [3]:
```
# 定数を定義
a = tf.constant(10)
print('a:', a)
# 変数を定義
x = tf.Variable(1)
print('x:', x)

calc_op = x * a

# xの値を更新
update_x = tf.assign(x, calc_op)

sess = tf.Session()

# 変数の初期化
init = tf.global_variables_initializer()
sess.run(init)

print(sess.run(x))

sess.run(update_x)
print(sess.run(x))

sess.run(update_x)
print(sess.run(x))
```

```
a: Tensor("Const_3:0", shape=(), dtype=int32)
x: <tf.Variable 'Variable:0' shape=() dtype=int32_ref>
1
10
100
```

・深層学習４－３～実装演習
線形回帰

線形回帰

[try]

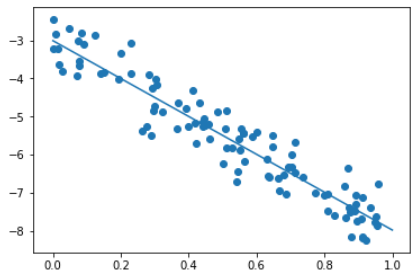
- noiseの値を変更しよう
- dの数値を変更しよう

In [9]:
```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
%matplotlib inline

iters_num = 300
plot_interval = 10

# データを生成
n = 100
x = np.random.rand(n)
#d = 3 * x + 2
d = -5 * x - 3

# ノイズを加える
#noise = 0.3
noise = 0.5
d = d + noise * np.random.randn(n)

# 入力値
xt = tf.placeholder(tf.float32)
dt = tf.placeholder(tf.float32)

# 最適化の対象の変数を初期化
W = tf.Variable(tf.zeros([1]))
b = tf.Variable(tf.zeros([1]))

y = W * xt + b

# 誤差関数 平均2乗誤差
loss = tf.reduce_mean(tf.square(y - dt))
optimizer = tf.train.GradientDescentOptimizer(0.1)
train = optimizer.minimize(loss)

# 初期化
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

# 作成したデータをトレーニングデータとして準備
x_train = x.reshape(-1,1)
d_train = d.reshape(-1,1)
```

```
# トレーニング
for i in range(iters_num):
    sess.run(train, feed_dict={xt:x_train,dt:d_train})
    if (i+1) % plot_interval == 0:
        loss_val = sess.run(loss, feed_dict={xt:x_train,dt:d_train})
        W_val = sess.run(W)
        b_val = sess.run(b)
        print('Generation: ' + str(i+1) + '. 誤差 = ' + str(loss_val))

print(W_val)
print(b_val)

# 予測関数
def predict(x):
    return W_val * x + b_val

fig = plt.figure()
subplot = fig.add_subplot(1, 1, 1)
plt.scatter(x, d)
linex = np.linspace(0, 1, 2)
liney = predict(linex)
subplot.plot(linex,liney)
plt.show()
```

```
Generation: 10. 誤差 = 0.7482822
Generation: 20. 誤差 = 0.56392235
Generation: 30. 誤差 = 0.48737285
Generation: 40. 誤差 = 0.42837664
Generation: 50. 誤差 = 0.38277528
Generation: 60. 誤差 = 0.34752724
Generation: 70. 誤差 = 0.32028186
Generation: 80. 誤差 = 0.2992222
Generation: 90. 誤差 = 0.28294402
Generation: 100. 誤差 = 0.2703613
Generation: 110. 誤差 = 0.26063555
Generation: 120. 誤差 = 0.25311792
Generation: 130. 誤差 = 0.24730708
```

```
Generation: 140. 誤差 = 0.2428155
Generation: 150. 誤差 = 0.2393437
Generation: 160. 誤差 = 0.23666015
Generation: 170. 誤差 = 0.23458584
Generation: 180. 誤差 = 0.23298246
Generation: 190. 誤差 = 0.23174316
Generation: 200. 誤差 = 0.23078525
Generation: 210. 誤差 = 0.2300448
Generation: 220. 誤差 = 0.22947244
Generation: 230. 誤差 = 0.22903
Generation: 240. 誤差 = 0.22868806
Generation: 250. 誤差 = 0.22842377
Generation: 260. 誤差 = 0.22821946
Generation: 270. 誤差 = 0.22806156
Generation: 280. 誤差 = 0.22793946
Generation: 290. 誤差 = 0.2278451
Generation: 300. 誤差 = 0.22777212
[-4.9749136]
[-3.0087068]
```



・深層学習４－４～実装演習
　非線形回帰（noise、dの変更）

非線形回帰

[try]

- noiseの値を変更しよう
- dの数値を変更しよう

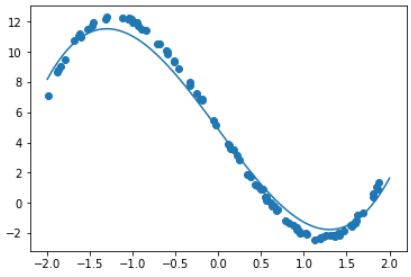
In [13]:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

iters_num = 10000
plot_interval = 100

```

```python
#データを生成
n=100
x = np.random.rand(n).astype(np.float32) * 4 - 2
#d =  - 0.4 * x ** 3 + 1.6 * x ** 2 - 2.8 * x + 1
d =  2 * x ** 3 + 0 * x ** 2 - 9 * x + 5

#  ノイズを加える
#noise = 0.05
noise = 0.1
d = d + noise * np.random.randn(n)

#モデル
#bを使っていないことに注意
xt = tf.placeholder(tf.float32, [None, 4])
dt = tf.placeholder(tf.float32, [None, 1])
W = tf.Variable(tf.random_normal([4, 1], stddev=0.01))
y = tf.matmul(xt,W)

#誤差関数 平均２乗誤差
loss = tf.reduce_mean(tf.square(y - dt))
optimizer = tf.train.AdamOptimizer(0.001)
train = optimizer.minimize(loss)

#初期化
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

#作成したデータをトレーニングデータとして準備
d_train = d.reshape(-1,1)
x_train = np.zeros([n, 4])
for i in range(n):
    for j in range(4):
        x_train[i, j] = x[i]**j

```

```python
# トレーニング
for i in range(iters_num):
    if (i+1) % plot_interval == 0:
        loss_val = sess.run(loss, feed_dict={xt:x_train, dt:d_train})
        W_val = sess.run(W)
        print('Generation: ' + str(i+1) + '. 誤差 = ' + str(loss_val))
    sess.run(train, feed_dict={xt:x_train,dt:d_train})

print(W_val[::-1])

# 予測関数
def predict(x):
    result = 0.
    for i in range(0,4):
        result += W_val[i,0] * x ** i
    return result

fig = plt.figure()
subplot = fig.add_subplot(1,1,1)
plt.scatter(x ,d)
linex = np.linspace(-2,2,100)
liney = predict(linex)
subplot.plot(linex,liney)
plt.show()
```

```
[[ 1.5053756 ]
 [ 0.01290111]
 [-7.6623034 ]
 [ 4.8614435 ]]
```



・深層学習４－５～実装演習
非線形回帰（モデルを y=30x2+0.5x+2 に変更、iters_num、learning_rate の調整）

非線形回帰

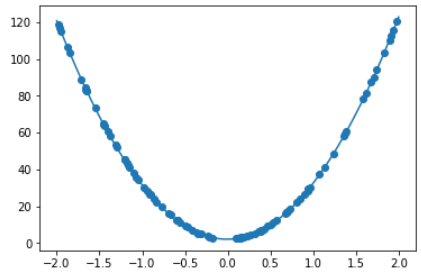
[try]

- noiseの値を変更しよう
- dの数値を変更しよう

In [18]:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

iters_num = 50000
plot_interval = 500

# データを生成
n=100
x = np.random.rand(n).astype(np.float32) * 4 - 2
#d =  - 0.4 * x ** 3 + 1.6 * x ** 2 - 2.8 * x + 1
#d =  2 * x ** 3 + 0 * x ** 2 - 9 * x + 5
d =  30 * x ** 2 + 0.5 * x + 2

# ノイズを加える
#noise = 0.05
noise = 0.1
d = d + noise * np.random.randn(n)

# モデル
# bを使っていないことに注意.
#xt = tf.placeholder(tf.float32, [None, 4])
xt = tf.placeholder(tf.float32, [None, 3])
dt = tf.placeholder(tf.float32, [None, 1])
#W = tf.Variable(tf.random_normal([4, 1], stddev=0.01))
W = tf.Variable(tf.random_normal([3, 1], stddev=0.01))
y = tf.matmul(xt,W)

# 誤差関数 平均２乗誤差
loss = tf.reduce_mean(tf.square(y - dt))
#optimizer = tf.train.AdamOptimizer(0.001)
optimizer = tf.train.AdamOptimizer(0.01)
train = optimizer.minimize(loss)

# 初期化
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

# 作成したデータをトレーニングデータとして準備
d_train = d.reshape(-1,1)
#x_train = np.zeros([n, 4])
x_train = np.zeros([n, 3])
for i in range(n):
#    for j in range(4):
    for j in range(3):
        x_train[i, j] = x[i]**j

# トレーニング
for i in range(iters_num):
    if (i+1) % plot_interval == 0:
        loss_val = sess.run(loss, feed_dict={xt:x_train, dt:d_train})
        W_val = sess.run(W)
        print('Generation: ' + str(i+1) + '. 誤差 = ' + str(loss_val))
    sess.run(train, feed_dict={xt:x_train,dt:d_train})

print(W_val[::-1])

# 予測関数
def predict(x):
    result = 0.
#    for i in range(0,4):
    for i in range(0,3):
        result += W_val[i,0] * x ** i
    return result

fig = plt.figure()
subplot = fig.add_subplot(1,1,1)
plt.scatter(x ,d)
linex = np.linspace(-2,2,100)
liney = predict(linex)
subplot.plot(linex,liney)
plt.show()
```

```
Generation: 50000. 誤差 = 0.01098307
[[29.9922  ]
 [ 0.50275916]
 [ 2.0037656 ]]
```



・深層学習４－６～実装演習
分類１層（mnist）

分類1層 (mnist)

[try]

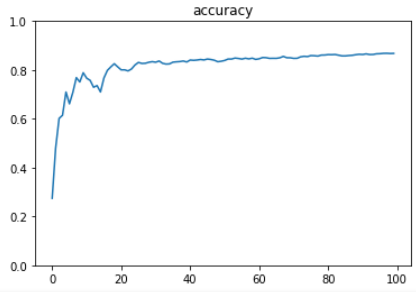
- x：入力値, d：教師データ, W：重み, b：バイアス をそれぞれ定義しよう

In [19]:

```
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

iters_num = 100
batch_size = 100
plot_interval = 1

# ---------- ここを補填 ----------
x = tf.placeholder(tf.float32, [None, 784])
d = tf.placeholder(tf.float32, [None, 10])
W = tf.Variable(tf.random_normal([784, 10], stddev=0.01))
b =  tf.Variable(tf.zeros(10))
# --------------------------------------------------

y = tf.nn.softmax(tf.matmul(x, W) + b)

# 交差エントロピー
cross_entropy = -tf.reduce_sum(d * tf.log(y), reduction_indices=[1])
loss = tf.reduce_mean(cross_entropy)
train = tf.train.GradientDescentOptimizer(0.1).minimize(loss)

# 正誤を保存
correct = tf.equal(tf.argmax(y, 1), tf.argmax(d, 1))
# 正解率
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

```

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

accuracies = []
for i in range(iters_num):
    x_batch, d_batch = mnist.train.next_batch(batch_size)
    sess.run(train, feed_dict={x: x_batch, d: d_batch})
    if (i+1) % plot_interval == 0:
        print(sess.run(correct, feed_dict={x: mnist.test.images, d: mnist.test.labels}))
        accuracy_val = sess.run(accuracy, feed_dict={x: mnist.test.images, d: mnist.test.labels})
        accuracies.append(accuracy_val)
        print('Generation: ' + str(i+1) + '. 正解率 = ' + str(accuracy_val))


lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies)
plt.title("accuracy")
plt.ylim(0, 1.0)
plt.show()
```

```
Generation: 99. 正解率 = 0.8675
[ True  True  True ...  True False  True]
Generation: 100. 正解率 = 0.868
```



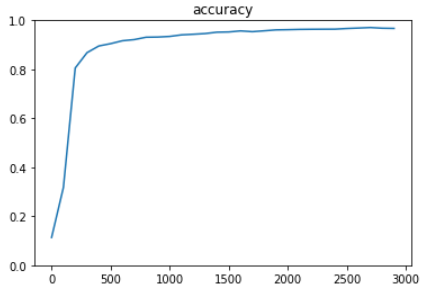
分類３層（mnist）

In [33]:

```python
import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
import matplotlib.pyplot as plt

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

iters_num = 3000
batch_size = 100
plot_interval = 100

#hidden_layer_size_1 = 600
#hidden_layer_size_2 = 300
hidden_layer_size_1 = 1000
hidden_layer_size_2 = 500

dropout_rate = 0.5

x = tf.placeholder(tf.float32, [None, 784])
d = tf.placeholder(tf.float32, [None, 10])
W1 = tf.Variable(tf.random_normal([784, hidden_layer_size_1], stddev=0.01))
W2 = tf.Variable(tf.random_normal([hidden_layer_size_1, hidden_layer_size_2], stddev=0.01))
W3 = tf.Variable(tf.random_normal([hidden_layer_size_2, 10], stddev=0.01))

b1 = tf.Variable(tf.zeros([hidden_layer_size_1]))
b2 = tf.Variable(tf.zeros([hidden_layer_size_2]))
b3 = tf.Variable(tf.zeros([10]))

z1 = tf.sigmoid(tf.matmul(x, W1) + b1)
z2 = tf.sigmoid(tf.matmul(z1, W2) + b2)

keep_prob = tf.placeholder(tf.float32)
drop = tf.nn.dropout(z2, keep_prob)

y = tf.nn.softmax(tf.matmul(drop, W3) + b3)
loss = tf.reduce_mean(-tf.reduce_sum(d * tf.log(y), reduction_indices=[1]))
```

```python
#optimizer = tf.train.AdamOptimizer(1e-4)
#optimizer = tf.train.MomentumOptimizer(0.1,0.9)
#optimizer = tf.train.GradientDescentOptimizer(0.5)
#optimizer = tf.train.AdagradOptimizer(0.1)
optimizer = tf.train.RMSPropOptimizer(0.001)

train = optimizer.minimize(loss)
correct = tf.equal(tf.argmax(y, 1), tf.argmax(d, 1))
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

accuracies = []
for i in range(iters_num):
    x_batch, d_batch = mnist.train.next_batch(batch_size)
    sess.run(train, feed_dict={x:x_batch, d:d_batch, keep_prob:(1 - dropout_rate)})
    if (i+1) % plot_interval == 0:
        accuracy_val = sess.run(accuracy, feed_dict={x:mnist.test.images, d:mnist.test.labels, keep_prob:1.0})
        accuracies.append(accuracy_val)
        print('Generation: ' + str(i+1) + '. 正解率 = ' + str(accuracy_val))

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies)
plt.title("accuracy")
plt.ylim(0, 1.0)
plt.show()
```

```
Generation: 2800. 正解率 = 0.9708
Generation: 2900. 正解率 = 0.968
Generation: 3000. 正解率 = 0.9672
```



・深層学習 4 － 7 ～実装演習

分類CNN (mnist)

conv - relu - pool - conv - relu - pool -
affin - relu - dropout - affin - softmax

[try]

- ドロップアウト率を0に変更しよう

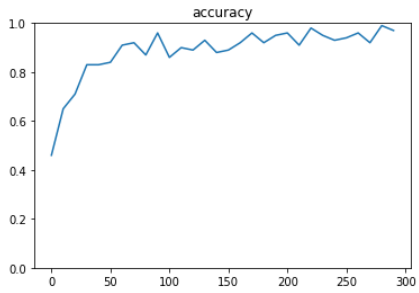
In [37]:

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
import matplotlib.pyplot as plt

iters_num = 300
batch_size = 100
plot_interval = 10

dropout_rate = 0.0

# placeholder
x = tf.placeholder(tf.float32, shape=[None, 784])
d = tf.placeholder(tf.float32, shape=[None, 10])

# 画像を784の一次元から28x28の二次元に変換する
# 画像を28x28にreshape
x_image = tf.reshape(x, [-1,28,28,1])

# 第一層のweightsとbiasのvariable
W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))
b_conv1 = tf.Variable(tf.constant(0.1, shape=[32]))

```

```
# 第一層のconvolutionalとpool
# strides[0] = strides[3] = 1固定
h_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1, strides=[1, 1, 1, 1], padding='SAME') + b_conv1)
# プーリングサイズ n*n にしたい場合 ksize=[1, n, n, 1]
h_pool1 = tf.nn.max_pool(h_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

# 第二層
W_conv2 = tf.Variable(tf.truncated_normal([5, 5, 32, 64], stddev=0.1))
b_conv2 = tf.Variable(tf.constant(0.1, shape=[64]))
h_conv2 = tf.nn.relu(tf.nn.conv2d(h_pool1, W_conv2, strides=[1, 1, 1, 1], padding='SAME') + b_conv2)
h_pool2 = tf.nn.max_pool(h_conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

# 第一層と第二層でreduceされてできた特徴に対してrelu
W_fc1 = tf.Variable(tf.truncated_normal([7 * 7 * 64, 1024], stddev=0.1))
b_fc1 = tf.Variable(tf.constant(0.1, shape=[1024]))
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

# Dropout
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

# 出来上がったものに対してSoftmax
W_fc2 = tf.Variable(tf.truncated_normal([1024, 10], stddev=0.1))
b_fc2 = tf.Variable(tf.constant(0.1, shape=[10]))
y_conv=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)

# 交差エントロピー
loss = -tf.reduce_sum(d * tf.log(y_conv))

train = tf.train.AdamOptimizer(1e-4).minimize(loss)

correct = tf.equal(tf.argmax(y_conv,1), tf.argmax(d,1))
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

```

```
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)


accuracies = []
for i in range(iters_num):
    x_batch, d_batch = mnist.train.next_batch(batch_size)
    sess.run(train, feed_dict={x: x_batch, d: d_batch, keep_prob: 1-dropout_rate})
    if (i+1) % plot_interval == 0:
        accuracy_val = sess.run(accuracy, feed_dict={x:x_batch, d: d_batch, keep_prob: 1.0})
        accuracies.append(accuracy_val)
        print('Generation: ' + str(i+1) + '. 正解率 = ' + str(accuracy_val))


lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies)
plt.title("accuracy")
plt.ylim(0, 1.0)
plt.show()
```

```
Generation: 290. 正解率 = 0.99
Generation: 300. 正解率 = 0.97
```



・深層学習４－１３～実装演習
－初期値でモデルの精度が変わる
→精度向上が必要な場面では、初期値のパターンを複数試しそれらを平均することもある
－基本的に、学習回数を増やすことで精度は上がる
現場では汎化性能が重要
→学習データで学習しながら、学習に用いない検証用データで精度を確認し、精度が下がらなくなるところまで学習回数を増やすことが重要
－ＯＲ回路は非線形なデータセットのため、活性化関数がＳＩＧＭＯＩＤでは学習が進まない
→活性化関数をＲｅＬＵなどの非線形な活性化関数にすることで学習が進む
－バッチサイズを大きくすることで１エポックあたりの学習の回数を減らすことができ、学習速度が速くなる
－バッチサイズは２の倍数を想定する（ＧＰＵ使用時に性能が一番よく出るため）
－データ数が少ないときにはバッチサイズを少なめに、多いときはバッチサイズを多くする
（データ数が多いときにバッチサイズを少なくすると１エポックあたりの学習時間が大きくなるため）

np.random.seed(1)、エポック数 100、バッチサイズ 1、AND 回路

In [7]:

```
# モジュール読み込み
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD
%matplotlib inline

# 乱数を固定値で初期化
#np.random.seed(0)
np.random.seed(1)

# シグモイドの単純パーセプトロン作成
model = Sequential()
model.add(Dense(input_dim=2, units=1))
model.add(Activation('sigmoid'))
model.summary()

model.compile(loss='binary_crossentropy', optimizer=SGD(lr=0.1))

# トレーニング用入力 X と正解データ T
X = np.array( [[0,0], [0,1], [1,0], [1,1]] )
#T = np.array( [[0], [1], [1], [1]] )
T = np.array( [[0], [0], [0], [1]] )

# トレーニング
#model.fit(X, T, epochs=30, batch_size=1)
model.fit(X, T, epochs=100, batch_size=1)

# トレーニングの入力を流用して実際に分類
Y = model.predict_classes(X, batch_size=1)

print("TEST")
print(Y == T)
```

```
Epoch 94/100
4/4 [==============================] - 0s 3ms/step - loss: 0.2610
Epoch 95/100
4/4 [==============================] - 0s 3ms/step - loss: 0.2596
Epoch 96/100

4/4 [==============================] - 0s 3ms/step - loss: 0.2583
Epoch 97/100
4/4 [==============================] - 0s 3ms/step - loss: 0.2569
Epoch 98/100
4/4 [==============================] - 0s 3ms/step - loss: 0.2555
Epoch 99/100
4/4 [==============================] - 0s 3ms/step - loss: 0.2541
Epoch 100/100
4/4 [==============================] - 0s 3ms/step - loss: 0.2527
TEST
[[ True]
 [ True]
 [ True]
 [ True]]
```

np.random.seed(1)、エポック数 100、OR 回路、バッチサイズ 10

In [11]:

```
# モジュール読み込み
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD
%matplotlib inline

# 乱数を固定値で初期化
#np.random.seed(0)
np.random.seed(1)

# シグモイドの単純パーセプトロン作成
model = Sequential()
model.add(Dense(input_dim=2, units=1))
model.add(Activation('sigmoid'))
model.summary()

model.compile(loss='binary_crossentropy', optimizer=SGD(lr=0.1))

# トレーニング用入力 X と正解データ T
X = np.array( [[0,0], [0,1], [1,0], [1,1]] )
T = np.array( [[0], [1], [1], [1]] )
#T = np.array( [[0], [0], [0], [1]] )

# トレーニング
#model.fit(X, T, epochs=30, batch_size=1)
#model.fit(X, T, epochs=100, batch_size=1)
model.fit(X, T, epochs=100, batch_size=10)

# トレーニングの入力を流用して実際に分類
Y = model.predict_classes(X, batch_size=1)

print("TEST")
print(Y == T)
```

```
Epoch 94/100
4/4 [==============================] - 0s 750us/step - loss: 0.3112
Epoch 95/100
4/4 [==============================] - 0s 749us/step - loss: 0.3104

Epoch 96/100
4/4 [==============================] - 0s 750us/step - loss: 0.3096
Epoch 97/100
4/4 [==============================] - 0s 749us/step - loss: 0.3088
Epoch 98/100
4/4 [==============================] - 0s 3ms/step - loss: 0.3080
Epoch 99/100
4/4 [==============================] - 0s 749us/step - loss: 0.3071
Epoch 100/100
4/4 [==============================] - 0s 499us/step - loss: 0.3063
TEST
[[False]
 [ True]
 [ True]
 [ True]]
```

np.random.seed(1)、エポック数 300、OR 回路、バッチサイズ 10

In [12]:

```
# モジュール読み込み
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD
%matplotlib inline

# 乱数を固定値で初期化
#np.random.seed(0)
np.random.seed(1)

# シグモイドの単純パーセプトロン作成
model = Sequential()
model.add(Dense(input_dim=2, units=1))
model.add(Activation('sigmoid'))
model.summary()

model.compile(loss='binary_crossentropy', optimizer=SGD(lr=0.1))

# トレーニング用入力 X と正解データ T
X = np.array( [[0,0], [0,1], [1,0], [1,1]] )
T = np.array( [[0], [1], [1], [1]] )
#T = np.array( [[0], [0], [0], [1]] )

# トレーニング
#model.fit(X, T, epochs=30, batch_size=1)
#model.fit(X, T, epochs=100, batch_size=1)
#model.fit(X, T, epochs=100, batch_size=10)
model.fit(X, T, epochs=300, batch_size=10)

# トレーニングの入力を流用して実際に分類
Y = model.predict_classes(X, batch_size=1)

print("TEST")
print(Y == T)
```

```
Epoch 294/300
4/4 [==============================] - 0s 1ms/step - loss: 0.2028
Epoch 295/300
4/4 [==============================] - 0s 749us/step - loss: 0.2025
Epoch 296/300
4/4 [==============================] - 0s 499us/step - loss: 0.2021
Epoch 297/300
4/4 [==============================] - 0s 500us/step - loss: 0.2018

Epoch 298/300
4/4 [==============================] - 0s 500us/step - loss: 0.2014
Epoch 299/300
4/4 [==============================] - 0s 749us/step - loss: 0.2010
Epoch 300/300
4/4 [==============================] - 0s 500us/step - loss: 0.2007
TEST
[[ True]
 [ True]
 [ True]
 [ True]]
```

np.random.seed(1)、エポック数 300、OR 回路、バッチサイズ 10、ReLU

In [14]:

```
# モジュール読み込み
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD
%matplotlib inline

# 乱数を固定値で初期化
#np.random.seed(0)
np.random.seed(1)

# シグモイドの単純パーセプトロン作成
model = Sequential()
#model.add(Dense(input_dim=2, units=1))
model.add(Dense(input_dim=2, units=2,activation='relu'))
model.add(Dense(1))
model.add(Activation('sigmoid'))
model.summary()

model.compile(loss='binary_crossentropy', optimizer=SGD(lr=0.1))

# トレーニング用入力 X と正解データ T
X = np.array( [[0,0], [0,1], [1,0], [1,1]] )
T = np.array( [[0], [1], [1], [1]] )
#T = np.array( [[0], [0], [0], [1]] )

# トレーニング
#model.fit(X, T, epochs=30, batch_size=1)
#model.fit(X, T, epochs=100, batch_size=1)
#model.fit(X, T, epochs=100, batch_size=10)
model.fit(X, T, epochs=300, batch_size=10)

# トレーニングの入力を流用して実際に分類
Y = model.predict_classes(X, batch_size=1)

print("TEST")
print(Y == T)
```

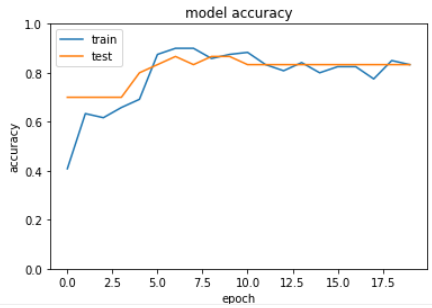
```
Epoch 294/300
4/4 [==============================] - 0s 507us/step - loss: 0.0786
Epoch 295/300
4/4 [==============================] - 0s 0us/step - loss: 0.0783
Epoch 296/300
4/4 [==============================] - 0s 2ms/step - loss: 0.0782
Epoch 297/300
4/4 [==============================] - 0s 0us/step - loss: 0.0778
Epoch 298/300
4/4 [==============================] - 0s 507us/step - loss: 0.0774
Epoch 299/300
4/4 [==============================] - 0s 0us/step - loss: 0.0770

Epoch 300/300
4/4 [==============================] - 0s 3ms/step - loss: 0.0769
TEST
[[ True]
 [ True]
 [ True]
 [ True]]
```

・深層学習４－１４～実装演習
中間層の活性関数を sigmoid に変更

In [19]:

```
import matplotlib.pyplot as plt
from sklearn import datasets
iris = datasets.load_iris()
x = iris.data
d = iris.target

# from sklearn.cross_validation import train_test_split
from sklearn.model_selection import train_test_split
x_train, x_test, d_train, d_test = train_test_split(x, d, test_size=0.2)

from keras.models import Sequential
from keras.layers import Dense, Activation
# from keras.optimizers import SGD

#モデルの設定
model = Sequential()
model.add(Dense(12, input_dim=4))
# model.add(Activation('relu'))
model.add(Activation('sigmoid'))
model.add(Dense(3, input_dim=12))
model.add(Activation('softmax'))
model.summary()

model.compile(optimizer='sgd', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

history = model.fit(x_train, d_train, batch_size=5, epochs=20, verbose=1, validation_data=(x_test, d_test))
loss = model.evaluate(x_test, d_test, verbose=0)

#Accuracy
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.ylim(0, 1.0)
plt.show()
```

```
Epoch 11/20
120/120 [==============================] - 0s 575us/step - loss: 0.9530 - acc: 0.8833 - val_loss: 0.9368 - val_acc: 0.8333
Epoch 12/20
120/120 [==============================] - 0s 466us/step - loss: 0.9403 - acc: 0.8333 - val_loss: 0.9221 - val_acc: 0.8333
Epoch 13/20
120/120 [==============================] - 0s 475us/step - loss: 0.9258 - acc: 0.8083 - val_loss: 0.9068 - val_acc: 0.8333
Epoch 14/20
120/120 [==============================] - 0s 491us/step - loss: 0.9128 - acc: 0.8417 - val_loss: 0.8912 - val_acc: 0.8333
Epoch 15/20
120/120 [==============================] - 0s 500us/step - loss: 0.8984 - acc: 0.8000 - val_loss: 0.8755 - val_acc: 0.8333
Epoch 16/20
120/120 [==============================] - 0s 575us/step - loss: 0.8839 - acc: 0.8250 - val_loss: 0.8614 - val_acc: 0.8333
Epoch 17/20
120/120 [==============================] - 0s 583us/step - loss: 0.8705 - acc: 0.8250 - val_loss: 0.8463 - val_acc: 0.8333
Epoch 18/20
120/120 [==============================] - 0s 766us/step - loss: 0.8565 - acc: 0.7750 - val_loss: 0.8327 - val_acc: 0.8333
Epoch 19/20
120/120 [==============================] - 0s 650us/step - loss: 0.8438 - acc: 0.8500 - val_loss: 0.8191 - val_acc: 0.8333
Epoch 20/20
120/120 [==============================] - 0s 491us/step - loss: 0.8325 - acc: 0.8333 - val_loss: 0.8046 - val_acc: 0.8333
```

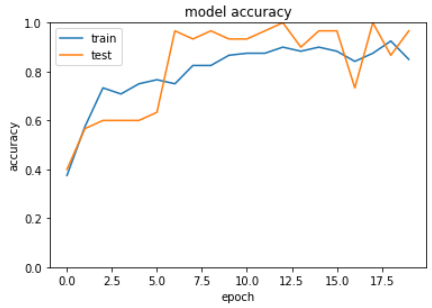


optimizer を SGD(lr=0.1)に変更
ー学習率を指定するため、ＳＧＤのインポートを行う。
　（そのままではデフォルトの lr=0.01 となる）

In [22]:

```
import matplotlib.pyplot as plt
from sklearn import datasets
iris = datasets.load_iris()
x = iris.data
d = iris.target

# from sklearn.cross_validation import train_test_split
from sklearn.model_selection import train_test_split
x_train, x_test, d_train, d_test = train_test_split(x, d, test_size=0.2)

from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD

#モデルの設定
model = Sequential()
model.add(Dense(12, input_dim=4))
# model.add(Activation('relu'))
model.add(Activation('sigmoid'))
model.add(Dense(3, input_dim=12))
model.add(Activation('softmax'))
model.summary()

# model.compile(optimizer='sgd', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.compile(optimizer=SGD(lr=0.1), loss='sparse_categorical_crossentropy', metrics=['accuracy'])

history = model.fit(x_train, d_train, batch_size=5, epochs=20, verbose=1, validation_data=(x_test, d_test))
loss = model.evaluate(x_test, d_test, verbose=0)

#Accuracy
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.ylim(0, 1.0)
plt.show()
```

```
Epoch 11/20
120/120 [==============================] - 0s 516us/step - loss: 0.4152 - acc: 0.8750 - val_loss: 0.3478 - val_acc: 0.9333
Epoch 12/20
120/120 [==============================] - 0s 450us/step - loss: 0.4026 - acc: 0.8750 - val_loss: 0.3525 - val_acc: 0.9667
Epoch 13/20
120/120 [==============================] - 0s 458us/step - loss: 0.3716 - acc: 0.9000 - val_loss: 0.3142 - val_acc: 1.0000
Epoch 14/20
120/120 [==============================] - 0s 625us/step - loss: 0.3800 - acc: 0.8833 - val_loss: 0.3379 - val_acc: 0.9000
Epoch 15/20
120/120 [==============================] - 0s 483us/step - loss: 0.3507 - acc: 0.9000 - val_loss: 0.2889 - val_acc: 0.9667
Epoch 16/20
120/120 [==============================] - 0s 575us/step - loss: 0.3368 - acc: 0.8833 - val_loss: 0.2764 - val_acc: 0.9667
Epoch 17/20
120/120 [==============================] - 0s 475us/step - loss: 0.3733 - acc: 0.8417 - val_loss: 0.3929 - val_acc: 0.7333
Epoch 18/20
120/120 [==============================] - 0s 475us/step - loss: 0.3072 - acc: 0.8750 - val_loss: 0.2376 - val_acc: 1.0000
Epoch 19/20
120/120 [==============================] - 0s 383us/step - loss: 0.2865 - acc: 0.9250 - val_loss: 0.3070 - val_acc: 0.8667
Epoch 20/20
120/120 [==============================] - 0s 650us/step - loss: 0.3087 - acc: 0.8500 - val_loss: 0.2286 - val_acc: 0.9667
```



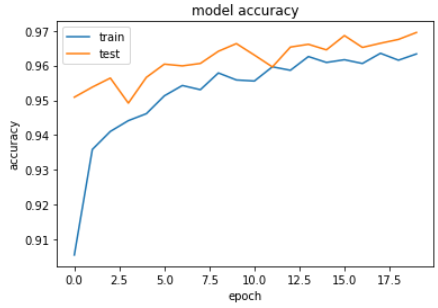
・深層学習４－１５～実装演習
　－誤差関数
　　ラベルが one_hot のときは categorical_crossentropy
　　one_hot でないときは sparse_categorical_crossentropy

load_mnist の one_hot_label を False に変更、誤差関数を sparse_categorical_crossentropy に変更、Adam の学習率を 0.01 に変更

In [24]:
```
# 必要なライブラリのインポート
import sys, os
sys.path.append(os.pardir)  # 親ディレクトリのファイルをインポートするための設定
import keras
import matplotlib.pyplot as plt
from data.mnist import load_mnist

#(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=False)

# 必要なライブラリのインポート、最適化手法はAdamを使う
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import Adam

# モデル作成
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
model.summary()

# バッチサイズ、エポック数
batch_size = 128
epochs = 20

#model.compile(loss='categorical_crossentropy',
#              optimizer=Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False),
#              metrics=['accuracy'])
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=Adam(lr=0.01, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False),
              metrics=['accuracy'])

history = model.fit(x_train, d_train, batch_size=batch_size, epochs=epochs, verbose=1, validation_data=(x_test, d_test))
loss = model.evaluate(x_test, d_test, verbose=0)
print('Test loss:', loss[0])
```

```
print('Test accuracy:', loss[1])
#Accuracy
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
# plt.ylim(0, 1.0)
plt.show()
```

```
Epoch 16/20
60000/60000 [==============================] - 26s 428us/step - loss: 0.1703 - acc: 0.9617 - val_loss: 0.1473 - val_acc: 0.9686
Epoch 17/20
60000/60000 [==============================] - 26s 430us/step - loss: 0.1748 - acc: 0.9606 - val_loss: 0.1801 - val_acc: 0.9652
Epoch 18/20
60000/60000 [==============================] - 25s 418us/step - loss: 0.1667 - acc: 0.9635 - val_loss: 0.1670 - val_acc: 0.9664
Epoch 19/20
60000/60000 [==============================] - 25s 420us/step - loss: 0.1629 - acc: 0.9615 - val_loss: 0.1525 - val_acc: 0.9675
Epoch 20/20
60000/60000 [==============================] - 25s 420us/step - loss: 0.1596 - acc: 0.9633 - val_loss: 0.1577 - val_acc: 0.9695
Test loss: 0.15772361929075523
Test accuracy: 0.9695
```



・深層学習４－１７～実装演習
　－出力ノード数を増やすと精度が上がる
　－ＤＲＯＰＯＵＴは汎化性能を上げるが学習の収束スピードが落ちていく
　－ｕｎｒｏｌｌ…ネットワークを展開するかどうかの指定
　　展開するとメモリ集中傾向となるがＲＮＮをスピードアップできる

RNN の出力ノード数 128、RNN の出力活性化関数 relu、RNN の入力 Dropout を 0.5 に設定、
RNN の再帰 Dropout を 0.3 に設定、RNN の unroll を True に設定

In [41]:

```
import sys, os
sys.path.append(os.pardir)  # 親ディレクトリのファイルをインポートするための設定
import numpy as np
import matplotlib.pyplot as plt

import keras
from keras.models import Sequential
from keras.layers.core import Dense, Dropout,Activation
from keras.layers.wrappers import TimeDistributed
from keras.optimizers import SGD
from keras.layers.recurrent import SimpleRNN, LSTM, GRU


# 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)[:, ::-1]


# A, B初期化 (a + b = d)
a_int = np.random.randint(largest_number/2, size=20000)
a_bin = binary[a_int] # binary encoding
b_int = np.random.randint(largest_number/2, size=20000)
b_bin = binary[b_int] # binary encoding

x_int = []
x_bin = []
for i in range(10000):
    x_int.append(np.array([a_int[i], b_int[i]]).T)
    x_bin.append(np.array([a_bin[i], b_bin[i]]).T)

x_int_test = []
x_bin_test = []
```

```
for i in range(10001, 20000):
    x_int_test.append(np.array([a_int[i], b_int[i]]).T)
    x_bin_test.append(np.array([a_bin[i], b_bin[i]]).T)

x_int = np.array(x_int)
x_bin = np.array(x_bin)
x_int_test = np.array(x_int_test)
x_bin_test = np.array(x_bin_test)


# 正解データ
d_int = a_int + b_int
d_bin = binary[d_int][0:10000]
d_bin_test = binary[d_int][10001:20000]

model = Sequential()

# model.add(SimpleRNN(units=16,
            # return_sequences=True,
            # input_shape=[8, 2],
            # go_backwards=False,
            # activation='relu',
            # dropout=0.5,
            # recurrent_dropout=0.3,
            # unroll = True,
model.add(SimpleRNN(units=128,
            return_sequences=True,
            input_shape=[8, 2],
            go_backwards=False,
            activation='relu',
            dropout=0.5,
            recurrent_dropout=0.3,
            unroll = True,
```

```
# model.add(LSTM(units=16,
#               # return_sequences=True,
#               # input_shape=[8, 2],
#           ))
# 出力層
model.add(Dense(1, activation='sigmoid', input_shape=(-1,2)))
model.summary()
# model.compile(loss='mean_squared_error', optimizer=SGD(lr=0.1), metrics=['accuracy'])
model.compile(loss='mse', optimizer='adam', metrics=['accuracy'])

history = model.fit(x_bin, d_bin.reshape(-1, 8, 1), epochs=5, batch_size=2)

# テスト結果出力
score = model.evaluate(x_bin_test, d_bin_test.reshape(-1,8,1), verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_11 (SimpleRNN)    (None, 8, 128)            16768
_________________________________________________________________
dense_47 (Dense)             (None, 8, 1)              129
=================================================================
Total params: 16,897
Trainable params: 16,897
Non-trainable params: 0
_________________________________________________________________
Epoch 1/5
10000/10000 [==============================] - 27s 3ms/step - loss: 0.2318 - acc: 0.5857
Epoch 2/5
10000/10000 [==============================] - 22s 2ms/step - loss: 0.2085 - acc: 0.6283
Epoch 3/5
10000/10000 [==============================] - 23s 2ms/step - loss: 0.2026 - acc: 0.6323
Epoch 4/5
10000/10000 [==============================] - 23s 2ms/step - loss: 0.2017 - acc: 0.6327
Epoch 5/5
10000/10000 [==============================] - 27s 3ms/step - loss: 0.1998 - acc: 0.6324
Test loss: 0.22181280599896794
Test accuracy: 0.7334358435724363
```

LSTM

In [43]:

```
import sys, os
sys.path.append(os.pardir)  # 親ディレクトリのファイルをインポートするための設定
import numpy as np
import matplotlib.pyplot as plt

import keras
from keras.models import Sequential
from keras.layers.core import Dense, Dropout,Activation
from keras.layers.wrappers import TimeDistributed
from keras.optimizers import SGD
from keras.layers.recurrent import SimpleRNN, LSTM, GRU


# 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)[:, ::-1]


# A, B初期化 (a + b = d)
a_int = np.random.randint(largest_number/2, size=20000)
a_bin = binary[a_int] # binary encoding
b_int = np.random.randint(largest_number/2, size=20000)
b_bin = binary[b_int] # binary encoding

x_int = []
x_bin = []
for i in range(10000):
    x_int.append(np.array([a_int[i], b_int[i]]).T)
    x_bin.append(np.array([a_bin[i], b_bin[i]]).T)

x_int_test = []
x_bin_test = []
```

```
for i in range(10001, 20000):
    x_int_test.append(np.array([a_int[i], b_int[i]]).T)
    x_bin_test.append(np.array([a_bin[i], b_bin[i]]).T)

x_int = np.array(x_int)
x_bin = np.array(x_bin)
x_int_test = np.array(x_int_test)
x_bin_test = np.array(x_bin_test)


# 正解データ
d_int = a_int + b_int
d_bin = binary[d_int][0:10000]
d_bin_test = binary[d_int][10001:20000]

model = Sequential()

# model.add(SimpleRNN(units=16,
            # return_sequences=True,
            # input_shape=[8, 2],
            # go_backwards=False,
            # activation='relu',
            # dropout=0.5,
            # recurrent_dropout=0.3,
            # unroll = True,
# model.add(SimpleRNN(units=128,
            # return_sequences=True,
            # input_shape=[8, 2],
            # go_backwards=False,
            # activation='relu',
            # dropout=0.5,
            # recurrent_dropout=0.3,
            # unroll = True,
model.add(LSTM(units=128,
            return_sequences=True,
            input_shape=[8, 2],
            dropout=0.5,
            recurrent_dropout=0.3,
            unroll = True,
          ))
# 出力層
model.add(Dense(1, activation='sigmoid', input_shape=(-1,2)))
model.summary()
# model.compile(loss='mean_squared_error', optimizer=SGD(lr=0.1), metrics=['accuracy'])
model.compile(loss='mse', optimizer='adam', metrics=['accuracy'])

history = model.fit(x_bin, d_bin.reshape(-1, 8, 1), epochs=5, batch_size=2)

# テスト結果出力
score = model.evaluate(x_bin_test, d_bin_test.reshape(-1,8,1), verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_3 (LSTM)                (None, 8, 128)            67072
_________________________________________________________________
dense_49 (Dense)             (None, 8, 1)              129
=================================================================
Total params: 67,201
Trainable params: 67,201
Non-trainable params: 0
_________________________________________________________________
Epoch 1/5
10000/10000 [==============================] - 69s 7ms/step - loss: 0.2380 - acc: 0.5742
Epoch 2/5
10000/10000 [==============================] - 70s 7ms/step - loss: 0.1796 - acc: 0.7223: 0s - loss: 0.1799 -
Epoch 3/5
10000/10000 [==============================] - 73s 7ms/step - loss: 0.1240 - acc: 0.8321
Epoch 4/5
10000/10000 [==============================] - 68s 7ms/step - loss: 0.0974 - acc: 0.8682: 1s - loss:
Epoch 5/5
10000/10000 [==============================] - 64s 6ms/step - loss: 0.0877 - acc: 0.8789
Test loss: 0.015369740964071562
Test accuracy: 0.993986898689869
```

２－１　強化学習とは
・長期的に報酬を最大化できるように環境の中で行動を選択できるエージェントを作ることを目標とする機械学習の一分野
　行動の結果として得られる利益をもとに行動を決定する原理を改善していく仕組み
２－２　強化学習の応用例
・環境　　：ゴルフコース
・エージェント：ピンまでの距離・風・ライ等の諸条件に基づいて次のショットの番手を選択
・行動　　：次のショットの番手を選択
・報酬　　：スコア
２－３　探索と利用のトレードオフ
・「環境について事前に完璧な知識を持っている」ことを前提としない
　→不完全な知識をもとにしながらデータを収集し、最適な行動を見つけていく
・過去のデータでベストとされる行動を常にとり続ける
　→ほかにベストな行動を見つけることができない（探索が足りない状態）
　未知の行動を常にとり続ける
　→過去の経験が生かせない（利用が足りない状態）
　探索と利用のトレードオフをうまく調整することが強化学習では重要となる
２－４　強化学習のイメージ
・エージェントがある方策を実施したときに環境はある状態ｓとなる
　（方策は方策関数で表される）
　エージェントは状態ｓを観測し、その状態ｓにおける報酬価値を受け取る
　（価値は高度価値関数で表される）
２－５　強化学習の差分
・教師あり学習・教師なし学習との違いは目標の違い
　教師あり学習であれば過去のデータから何かを予測すること、
　教師なし学習であればデータに含まれるパターンを見つけ出すことが目標
　強化学習は優れた方策を見つけ出すことが目標
２－６　行動価値関数
・価値を表す関数として、状態価値関数と行動価値関数の２種類がある
　ある状態の価値に注目する場合は状態価値関数
　状態と価値を組み合わせた価値に注目する場合は行動価値関数
　（Ｑ学習では行動価値関数を行動毎に更新することで学習を行う）
２－７　方策関数
・方策関数とは、方策ベースの強化学習手法においてある状態でどのような行動を取るかの確率を与える関数のこと
２－８　方策勾配法
・方策勾配法は、方策をモデル化して最適化する手法
　再帰的に更新するモデルとなっている

$$\theta^{(t+1)} = \theta^{(t)} + \epsilon \nabla J(\theta)$$

　ｊは方策の良さを表す。ｊの定義方法として以下がある。
　－平均報酬　：行動をとったときに生まれる価値の平均
　－割引報酬和：直近のものが重みづけが重く、過去になればなるほど減衰する
・方策勾配定理

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[(\nabla_\theta log \pi_\theta(a \mid s) Q^\pi(s,a))]$$

　上記は下記の２式から導出される

状態価値関数ｖ（ｓ）＝ｓｕｍ＿ａ（π（ａ｜ｓ）Ｑ（ｓ・ａ））
ベルマン方程式Ｑ（ｓ・ａ）＝ｓｕｍ＿ｓ´（Ｐ（ｓ´｜ｓ・ａ）［ｒ（ｓ・ａ・ｓ´）＋ｙＶ（ｓ´）］
　導出方法はＧｏｏｇｌｅで…