

## 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」

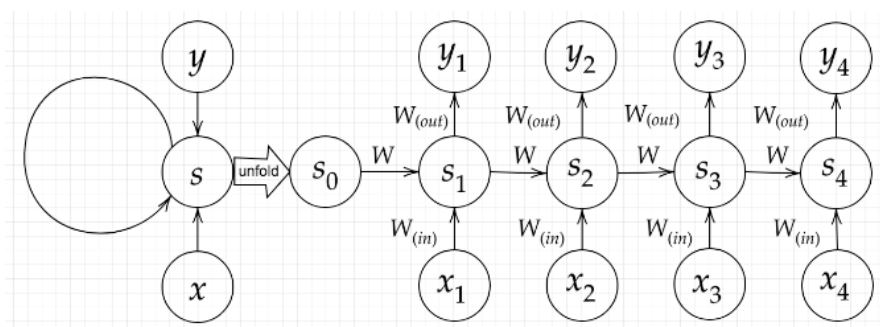
- ・ RNN：時系列データに対応したNN。音声データ、テキストデータ等を扱う。  
時系列データ～時間的順序を追って一定間隔で展開されるデータ。  
中間層、隠れ層に特長がある。  
次の層にデータが渡されてyが出力され、次の層へデータが渡されてゆく。

$$\begin{aligned}u^t &= W_{(in)}x^t + W z^{t-1} + b \\z^t &= f(W_{(in)}x^t + W z^{t-1} + b) \\v^t &= W_{(out)}z^t + c \\y^t &= g(W_{(out)}z^t + c)\end{aligned}$$

実装(IN)： `u[:,t+1] = np.dot(X, W_in) + np.dot(z[:,t].reshape(1, -1), W)`  
`z[:,t+1] = functions.relu(u[:,t+1])`

実装(OUT)： `y[:,t] = functions.sigmoid(np.dot(z[:,t+1].reshape(1, -1), W_out))`

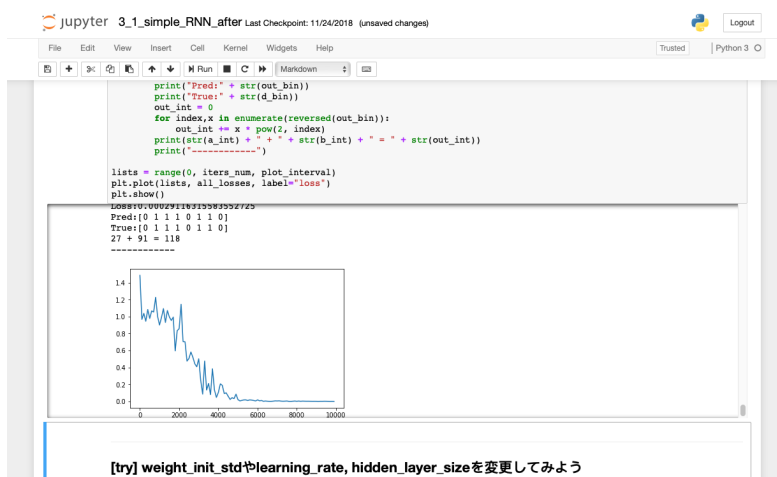
- ・ 確認テスト：RNNの3つの重み
  1. 入力から現在の中間層を定義するときの重み
  2. 中間層から出力を定義するときの重み
  3. 中間層から中間層へ渡されるとき重み
- ・ RNNの特長：時系列モデルを扱うには初期の状態と過去の時間t-1の状態を保持し、そこから次の時間のtを再帰的に求める再帰構造。



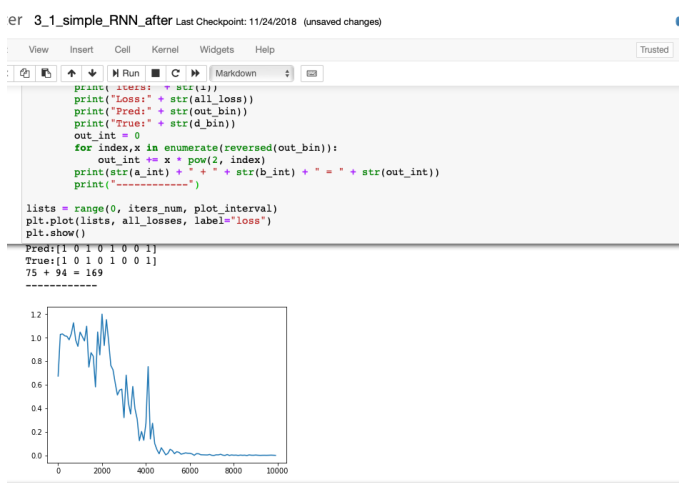
- ・演習1 : hidden\_layer\_size=10, learning\_rate=0.05で実行



- ・演習2：重みの初期化の変更



He



Initial weight, initial learning rate, hidden layer size, and activation function.

Xavier

・ 演習 3：中間層活性化関数

jupyter 3\_1\_simple\_RNN Last Checkpoint: 11/24/2018 (unsaved changes)

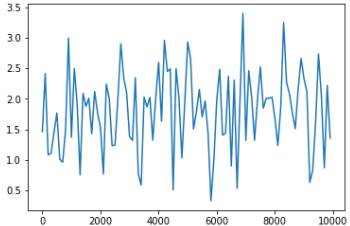
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

plt.show()

Pred:[1 1 0 1 0 1 1 0]

True:[1 0 0 1 1 1 0 0]

110 + 46 = 214



**[try] weight\_init\_stdやlearning\_rate, hidden\_layer\_sizeを変更してみよう**

**[try] 重みの初期化方法を変更してみよう**

Xavier, He

**[try] 中間層の活性化関数を変更してみよう**

ReLU(勾配爆発を確認しよう)

tanh(numpyにtanhが用意されている。導関数をd\_tanhとして作成しよう)

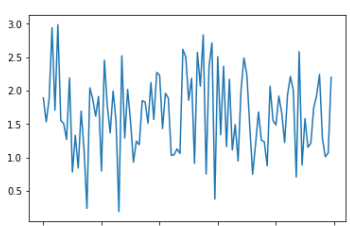
ReLU

jupyter 3\_1\_simple\_RNN Last Checkpoint: 11/24/2018 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

True:[0 1 1 0 0 0 1 1]

72 + 27 = 144



**[try] weight\_init\_stdやlearning\_rate, hidden\_layer\_sizeを変更してみよう**

**[try] 重みの初期化方法を変更してみよう**

Xavier, He

**[try] 中間層の活性化関数を変更してみよう**

ReLU(勾配爆発を確認しよう)

tanh(numpyにtanhが用意されている。導関数をd\_tanhとして作成しよう)

than

- ・ BPTT：誤差逆伝播の一種
- ・ 誤差逆伝播の復習：微分を逆算することで不要な再帰的計算を避けて微分する。
- ・ 確認テスト：チェーンルールを使用してdz/dxを求めよ。

$$z = t^2, t = x + y$$

$$dz/dx = dz/dt * dt/dx = 2t, dt/dx = 2(x + 1)$$

- ・ 数学的記述～チェーンルールを使用して誤差をvで微分、uをwで微分する。転置が必要。

実装：np.dot(X.T, delta[:,t].reshape(1,-1))

np.dot(z[:,t+1].reshape(-1,1), delta\_out[:,t].reshape(-1,1))

np.dot(z[:,t].reshape(-1,1), delta[:,t].reshape(1,-1))

バイアスの微分はサンプルでは割愛

$$\begin{aligned} u^t &= W_{(in)}x^t + W z^{t-1} + b & u[:,t+1] &= \text{np.dot}(X, W\_in) + \text{np.dot}(z[:,t].\text{reshape}(1, -1), W) \\ z^t &= f(W_{(in)}x^t + W z^{t-1} + b) & z[:,t+1] &= \text{functions.sigmoid}(u[:,t+1]) \\ v^t &= W_{(out)}z^t + c & \text{np.dot}(z[:,t+1].\text{reshape}(1, -1), W\_out) \\ y^t &= g(W_{(out)}z^t + c) & y[:,t] &= \text{functions.sigmoid}(\text{np.dot}(z[:,t+1].\text{reshape}(1, -1), W\_out)) \end{aligned}$$

- ・ 確認テスト：  $z1 = \text{sigmoid}(S0W + xiW(in) + c)$   
 $y1 = \text{sigmoid}(z1W(out) + c)$
- ・ BPTT全体像～ 今のWを前のWの中身と差分をひいて誤差パラメータを更新する。

$$\begin{aligned} W_{(in)}^{t+1} &= W_{(in)}^t - \epsilon \frac{\partial E}{\partial W_{(in)}} = W_{(in)}^t - \epsilon \sum_{z=0}^{T_t} \delta^{t-z} [x^{t-z}]^T & W\_in &-= \text{learning\_rate} * W\_in\_grad \\ W_{(out)}^{t+1} &= W_{(out)}^t - \epsilon \frac{\partial E}{\partial W_{(out)}} = W_{(out)}^t - \epsilon \delta^{out,t} [z^t]^T & W\_out &-= \text{learning\_rate} * W\_out\_grad \\ W^{t+1} &= W^t - \epsilon \frac{\partial E}{\partial W} = W_{(in)}^t - \epsilon \sum_{z=0}^{T_t} \delta^{t-z} [z^{t-z-1}]^T & W &-= \text{learning\_rate} * W\_grad \\ b^{t+1} &= b^t - \epsilon \frac{\partial E}{\partial b} = b^t - \epsilon \sum_{z=0}^{T_t} \delta^{t-z} \\ c^{t+1} &= c^t - \epsilon \frac{\partial E}{\partial c} = c^t - \epsilon \delta^{out,t} \end{aligned}$$

- ・ 演習チャレンジ：過去の中間層出力に依存する。過去の微分の値が常にかけている。

正解2 delta\_t.dot(U)

## 「Section2：LSTM」

- ・ RNNの課題：時系列を遡れば遡るほど勾配が消失してゆく。長い時系列の学習が困難。
- ・ 解決策：構造を変える。> LSTM
- ・ 勾配消失とは？ 勾配が学習が進むにつれ緩やかになり、パラメータの更新が殆ど行われず学習が進まなくなること。

- ・シグモイド関数：大きな値では出力の変化が微小なため、勾配消失を引き起こす
- ・確認テスト：シグモイド関数を入力値 0 で微分したときの値

$$f'(0) = (1 - \text{sigmoid}(0)) * \text{sigmoid}(0) = 0.5 * 0.5 = 0.25$$

- ・勾配爆発：勾配が層を逆伝播すること指数関数的に大きくなってゆくこと
- ・演習チャレンジ：勾配クリッピング～ノルムが閾値を超えたら正規化する

勾配 × 閾値が正解なので  $\text{gradient} * \text{rate}$  が正解

- ・CEC：LSTMの勾配爆発消失の解決方法。勾配を一律で 1 にして計算。

$$\delta^{t-z-1} = \delta^{t-z} \{ W f'(u^{t-z-1}) \} = 1$$

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なので、

学習特性がなくなってゆく。入力重み衝突、出力重み衝突が起こる。

- ・入力ゲート、出力ゲートを追加し、W、Uの重み行列をそれぞれに追加することで、値が変化し、課題が解決できる。
- ・忘却ゲート：CECは過去の情報が全て保管されていて、いらなくなった時に削除できない。  
いらなくなった過去の情報をそのタイミングで忘却するのが忘却ゲート。
- ・確認テスト：忘却ゲートが作用
- ・演習チャレンジ：忘却ゲート付きのLSTMの中間層の計算。1 ステップ前のセルの状態に入力ゲートと忘却ゲートを掛け合わせたもの。  
 $\text{input\_gate} * a + \text{forget\_gate} * c$
- ・覗き穴結合：CECの過去情報を任意のタイミングで他のノードに伝播させたり忘却させたいという課題を解決する方法。CEC自身の値に重み行列を介して伝播可能にする。

### 「Section3：GRU」

- ・GRU：LSTMはパラメータ数が多く計算負荷が高い。構造そのものを変えてたものがGRU。  
パラメータ数は減ったが精度は同等かそれ以上だが、LSTMの上位互換ではない。  
計算負荷がLSTMより低いモデル。更新ゲート、リセットゲートを持つ。  
リセットゲート～過去の隠れ状態をどれだけ無視するか。  
更新ゲート～隠れ状態を更新する。
- ・確認テスト：それぞれの課題  
LSTM～パラメータが多く計算負荷が高い。  
CEC～時系列を表現できず重みが一律。勾配が常に 1 で学習が進まなくなる。  
GRU～ゲートの数やパラメータの数が少なくより単純なモデル。
- ・演習チャレンジ：GRUの順伝播のコード、次の状態を計算  $r =$  リセットゲート、 $z =$  更新ゲート  
 $(1 - z) * h + z * \bar{h}$
- ・確認テスト：LSTMとGRUの違い

パラメータ数がLSTMは多く、GRUは少ない。  
構造が違う。LSTMは入力ゲート、忘却ゲート、出力ゲートからなり、  
GRUは更新ゲート、リセットゲートから成る。

#### 「Section4：双方向RNN」

- ・ 双方RNN：過去だけでなく未来の情報も加味して制度を向上させるモデル。  
時系列データを双方向から処理する。  
実用例としては文章の推敲や機械翻訳など。
- ・ 演習チャレンジ：双方向なので順伝播、逆伝播の両方の中間層表現を合わせたものが特徴量  
`np.concatenate([h_f, h_b[::-1]], axis=1)`    `concatenate`で配列の結合を行う

#### 「Section5：Seq2Seq」

- ・ Encoder / Decoderからなるモデル。自然言語処理、機械対話、翻訳等に使用。一問一答型。
- ・ Encoder RNN～入力データを単語のトークンに区切って渡す。ベクトルを順にRNNに入力。  
Taking～文章をトークンに分割。トークンごとのIDに分解。  
Embedding～IDからトークンを分散表現に変換。  
そ以後のベクターを入力したときのhidden stateをfinal stateとして保存。  
final stateがthought vectorと呼ばれ入力した文の意味ベクトルとなる。  
このベクトルがDecoderに渡される。
- ・ Decoder RNN～アウトプットデータを単語のトークンごとに生成する単語生成をする。  
final stateから各トークンの生成確率を出力。  
final state をinitial stateとして使用し、トークンを文字列に直す。  
Sampling～生成確率にもとづきtokenをランダムに選ぶ。  
Embedding～tokenをIDに変換  
Detokenize～initial state, sampling, embeddingを繰り返しtokenを文字列に  
直す。
- ・ 確認テスト：seq2seq (2)
- ・ 演習チャレンジ：one-hotベクトルwを単語埋め込みにより別の特徴量に変換する。  
(2)  $E \cdot \text{dot}(w)$
- ・ HRED～seq2seqは一問一答にしか対応できない。解決策としてVRED。  
問いに対して文脈もなくただ応答が行われる。  
過去のn-1個の発話から次の発話を生成する。より人間らしい文章を生成。  
seq2seq + Context RNNの構造。  
Context RNN～Encoderのまとめた書く文章の系列をまとめ、会話テキスト全体を表す

ベクトルに変換。過去の発話の履歴を加味できる。

課題：毎回同じような回答、短く情報に乏しい返答を返しがち。

- ・ VHRED～HREDにVAEの潜在変数の概念を追加することでHREDの課題を解決。
- ・ 確認テスト：違いを述べよ。

seq2seq～Encoder/Decoderからなり、一問一答に対応。

HRED～過去の発話も加味して返答を作成するが、短く同じ返答を作成しがち。

VHRED～HREDの課題を解決したもの

- ・ オートエンコーダー～教師なし学習。入力データのみ。

入力データを潜在変数Zに変換するEncoder NN、潜在変数を入力し元データを復元するDecoder NNからなる。次元削減ができる。

潜在変数Zはブラックボックスで何が入っているかわからない。

- ・ VAE～潜在変数Zに確率分布を仮定したモデル。
- ・ 確認テスト：答～確率分布

## 「Section6:Word2vec」

- ・ Word2vec：従来のRNNは固定長のデータしか扱えなかったが、可変長のデータを扱える。

学習データのセンテンスから重複のないボキャブラリを作成。

One-hotベクトル（0と1からなる配列）を入力に使用。

大規模データの分散表現が可能に。

ボキャブラリ数×任意の単語ベクトルの次元数で重み行列の表現が可能に。

次元を減らしても単語に意味のある重みをかけられる。

CBOWモデルとSkip-gramモデルの2つがあり、2つは真逆のアプローチをとる。

CBOW～コンテキスト（周辺の単語）からターゲット（中央の単語）を推測。

複数の入力層から一つの出力層を持つ。

Skip-gram～ターゲットからコンテキストを予測。

一つの入力層と複数の出力層を持つ。

## 「Section7:AttentionMechanism」

- ・ seq2seqは長い文章の対応が難しい。seq2seqは少ない単語数でも多い単語数でも固定次元のベクトルに入力しなければならない。文章が長くなる程、次元も大きくなっていく仕組みがAttentionメカニズム。入力と出力のどの単語が関連しているのかの関連度を学習する仕組み。現場ではseq2seq + Attentionの組み合わせで使うことが多い。
- ・ 確認テスト：違いを述べよ

word2vec～可変長ベクトルが扱える。意味のある重みの表現が可能。

seq2seq～一問一答型。Encoder / Decoderからなる。

Attention～入力と出力の単語を関連づけられる。

長い文章を入力しても翻訳が成り立つ。

- ・ 演習チャレンジ：隣接単語から表現ベクトルを作るのは、対象の単語の左右の単語を合わせたものに重みをかけて表現。

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



## 深層学習（後編2）

### 「Section1:Tensorflowの実装演習」

#### ・線形回帰

try: noise=0.1, d=2.5\*x + 2

noiseを大きくするとデータの点がまばらになって誤差が大きくなる。

noise=0.03にすると綺麗な一直線に。wを負の値にすると右下がりな直線に。

jupyter 4\_1\_tensorflow\_codes Last Checkpoint: 2018/11/24 (autosaved)



Logout

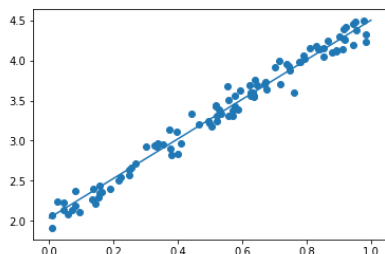
File Edit View Insert Cell Kernel Widgets Help

Trusted

Python 3

Run

```
Generation: 220. 誤差 = 0.011322469
Generation: 230. 誤差 = 0.011252583
Generation: 240. 誤差 = 0.011198301
Generation: 250. 誤差 = 0.011156134
Generation: 260. 誤差 = 0.011123402
Generation: 270. 誤差 = 0.011097968
Generation: 280. 誤差 = 0.011078221
Generation: 290. 誤差 = 0.011062885
Generation: 300. 誤差 = 0.011050973
[2.4599535]
[2.0390923]
```



#### 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

#### ・非線形回帰

try: 学習率=0.001, d = -1\*x^3 + x^2 -4x + 0.3

noise=0.5データのばらつきが大きくなるがうまく回帰している

noise=5ばらつきが大きくなり線が式を外れる

jupyter 4\_1\_tensorflow\_codes Last Checkpoint: 2018/11/24 (unsaved changes)



Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted

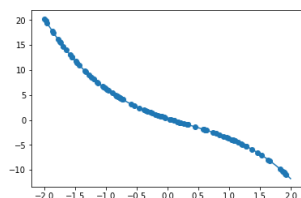
Python 3

Run

```
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()

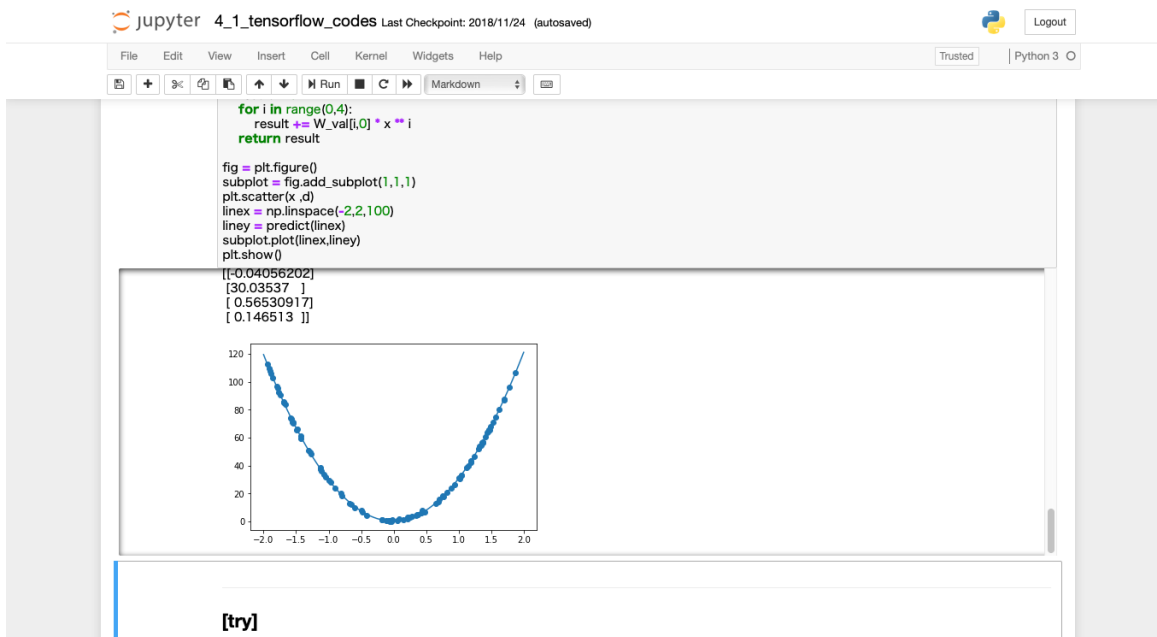
[[-1.0170567 ]
 [ 0.99846053]
 [-3.953313 ]
 [ 0.30287385]]
```



[try]

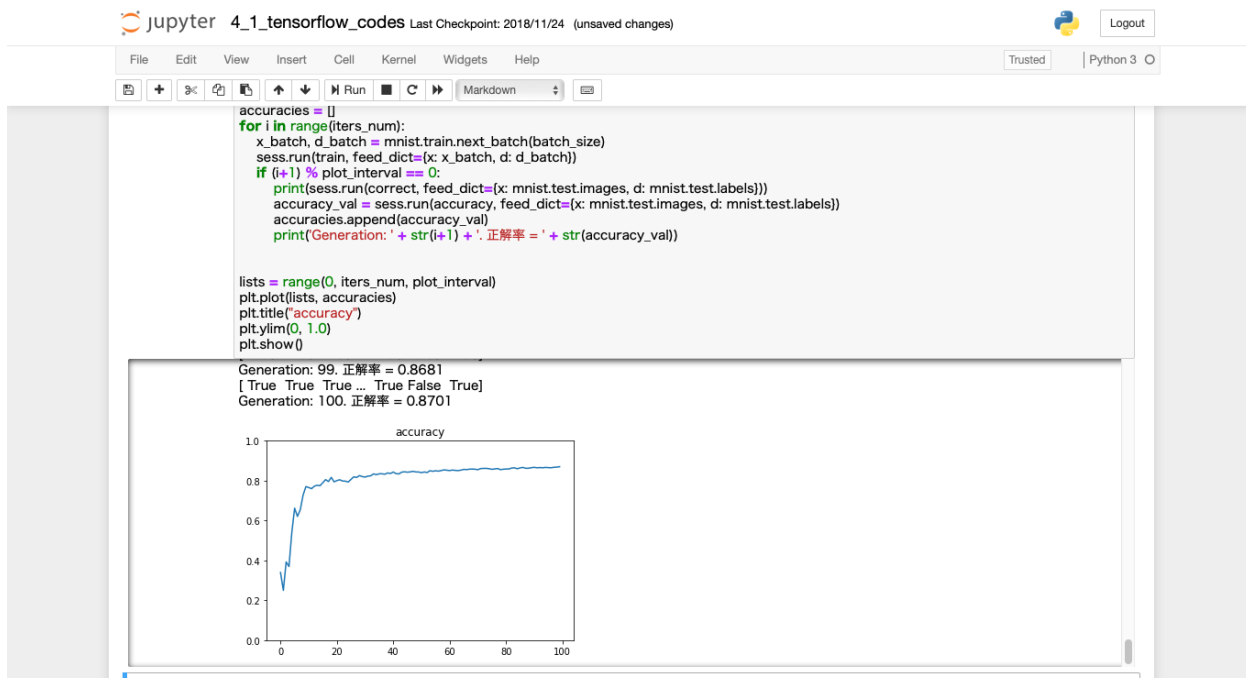
try: noise=0.5, 学習率=0.05

3万回実行するとかなり精度が上がる



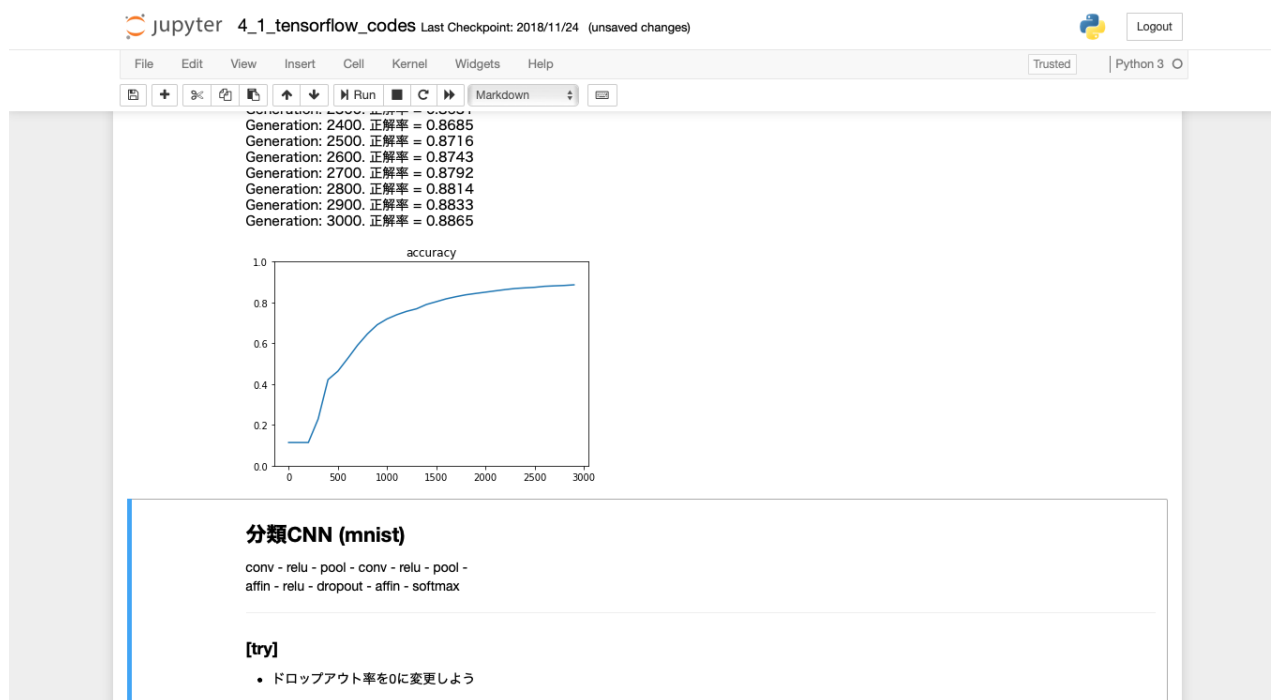
・ 分類一層mnist

try: x,d,w,bを定義



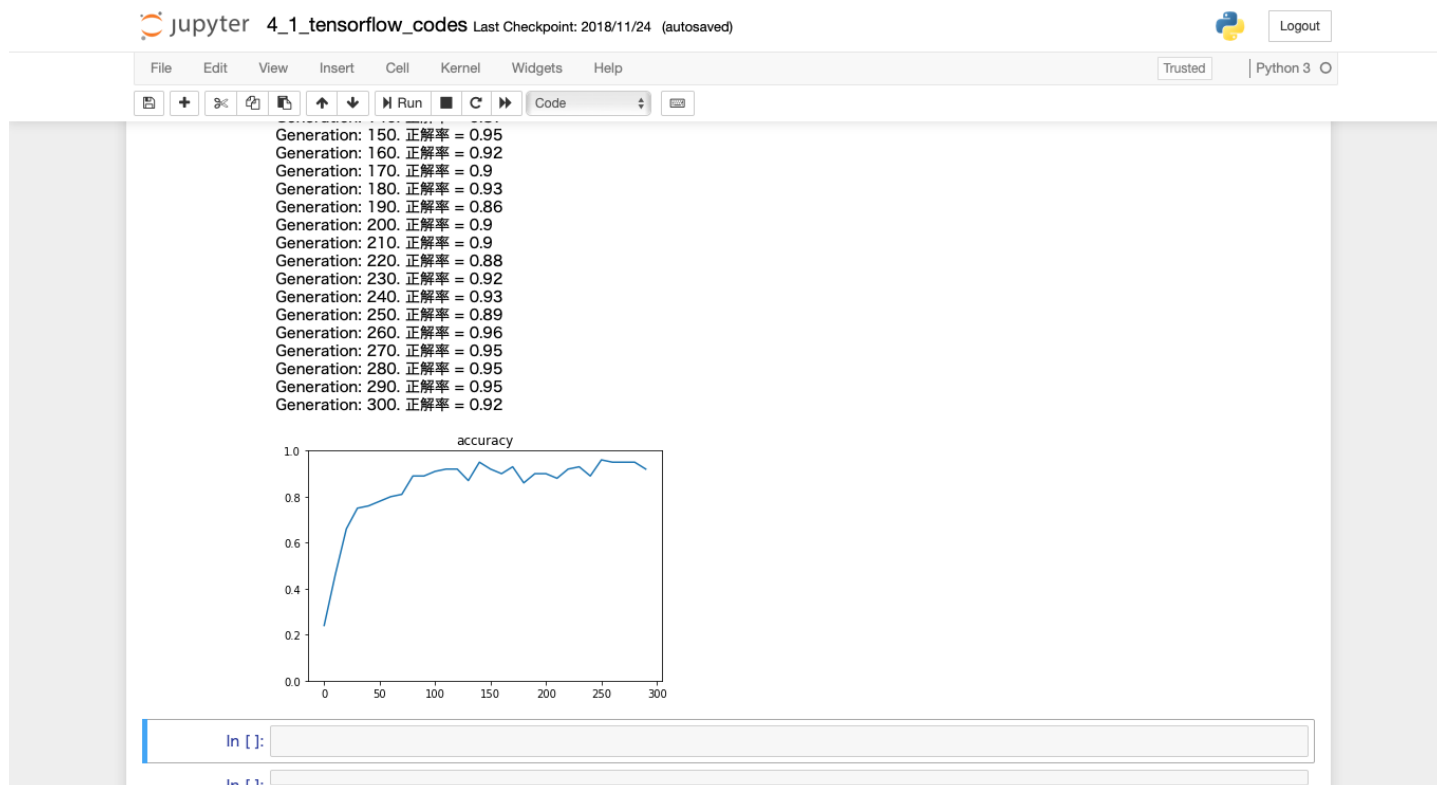
## ・ 分類 3 層mnist

try: hidden-layer1=500, optimizer=RMSProp



## ・ CNN

try: dropout=0 精度が落ちる



パラメータ数を減らすと学習は早く進むが精度は落ち、学習回数を増やせば精度は上がる  
tensorFlowでは畳み込み層がtf.cnn.conv2d, プーリング層をtf.cnn.max\_poolingで実装可能。

・ 例題 GoogLeNetのInception Module 答：a

小さい畳み込み層を積み重ねる。1×1の畳み込みフィルタで次元削減。  
小さなネットワークを1つのモジュールとして積み重ねる。

・ 例題 GoogLeNet Auxiliary Loss 答：a

Auxiliary Lossで計算量を抑えているわけではない。  
Auxiliary Lossは途中でクラス分岐している。  
Auxiliary Lossを使用しなくても、Batch Normalization で学習が進む。  
Auxiliary Lossを使用することでアンサンブル学習と同様の効果が。

・ 例題 ResNet(Residual Net) 152と圧倒的に深い層を持つモデル

他のモデルが深さを表現できなかったのは勾配消失問題を解決できなかったから（答：a）  
層をまたがる結合としてIdentity mapping（答：a）を使用。スキップコネクションの内側の層は残差を学習する。  
ブロックの入力にこれ以上の変換が必要ない場合は重みが0になり、小さい変換が求められる場合は対応する小さな変動をより見つけやすくすることで（答：a）層を深くしたときの勾配消失に対応した。

・ 例題 転移学習 答：a

ワンショット学習～学習に大量データを必要としない。  
移動しながら画像を認識するのではなく、一つのクラスに一つの画像を用意して特徴ベクトルを抽出し、特徴ベクトルの遠近を判断。  
学習データにバイアスが入っているとモデルがうまく機能しない。  
データの間違いに気づくのが大切なので（う）は正解。

・ 例題 物体検出 YOLO 答：a

YOLOは検出速度は速いがシンプルなモデルなので画像が重なっていると認識できない。  
b:候補領域は最大N個ではなく、必ず定義された数だけ。  
c:位置特定誤差と各クラスの確信度に対する2つの重みづけ。  
d:それぞれ2つの候補領域が出力される。

・ 確認テスト 違いを述べよ。

VGG～conv > conv > max poolingからなる単純なネットワークの積み重ね。パラメータが多い。  
GoogLeNet～Inception module, 1×1の畳み込み層で次元削減される。  
様々なサイズのフィルタを使用。  
ResNet～Skip connection, Identity moduleを使用し残差接続を行い、勾配消失にならずに

深い学習が可能に。

## ・ Keras

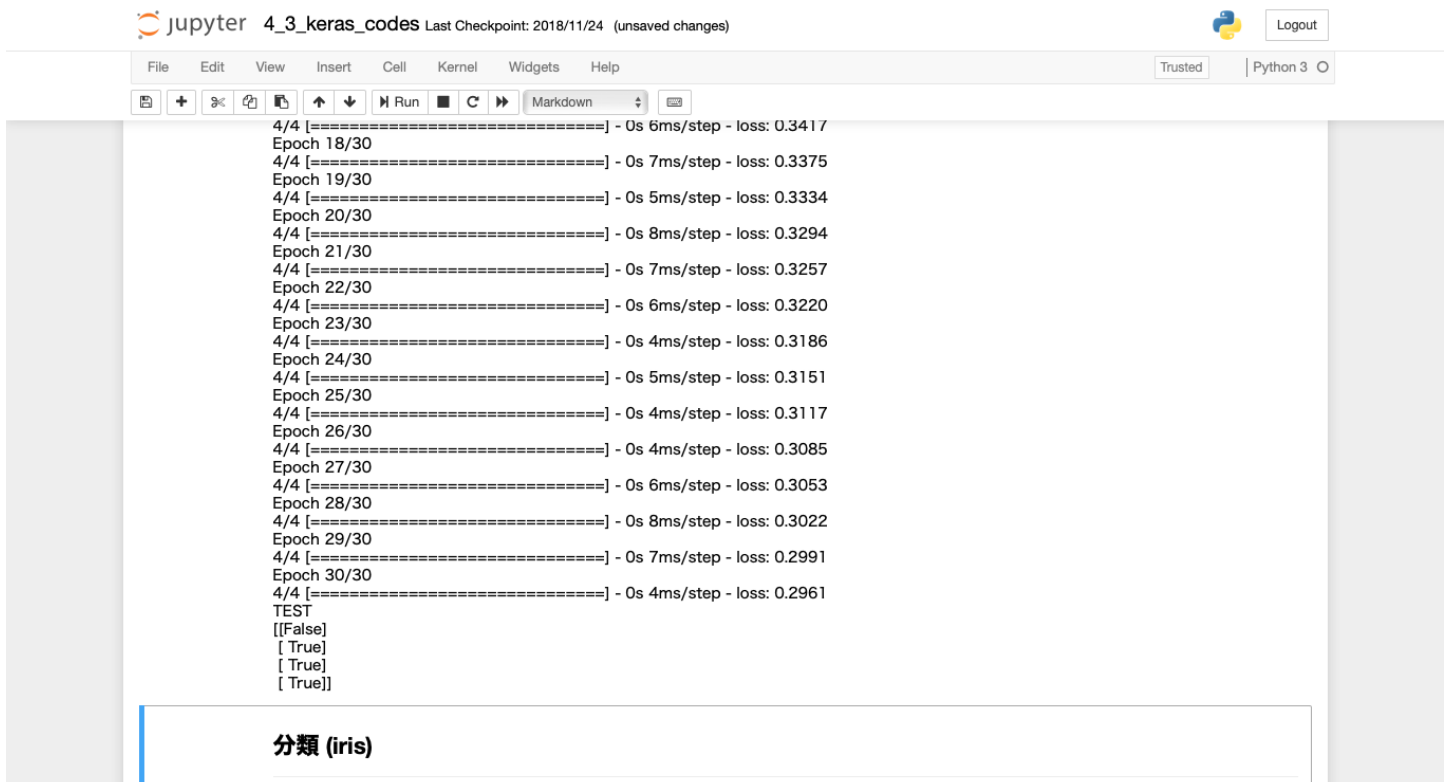
Kerasは簡単に実装できるが複雑なことはできないので、そのときはTensorFlowを利用。

try: 分類(iris) kerasはplaceholderを用意しなくて良い。



try:パーセプトロン

1. random\_seed(1) シードを変えると初期値が変わる



## 2. epoch=100 エポックが減ると学習時間が短くなり若干精度は下がる

jupyter4\_3\_keras\_codesLast Checkpoint: 2018/11/24 (autosaved)

FileEditViewInsertCellKernelWidgetsHelp

TrustedPython 3

Run

Markdown

```
# 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 6ms/step - loss: 0.1782
Epoch 95/100
4/4 [=====] - 0s 4ms/step - loss: 0.1770
Epoch 96/100
4/4 [=====] - 0s 5ms/step - loss: 0.1759
Epoch 97/100
4/4 [=====] - 0s 5ms/step - loss: 0.1748
Epoch 98/100
4/4 [=====] - 0s 7ms/step - loss: 0.1737
Epoch 99/100
4/4 [=====] - 0s 5ms/step - loss: 0.1726
Epoch 100/100
4/4 [=====] - 0s 8ms/step - loss: 0.1715
TEST
[[ True]
 [ True]
 [ True]
 [ True]]
```

### 分類 (iris)

[try]

- 中間層の活性化関数を sigmoid に変更しよう

## 3. AND回路 答えが変わる

jupyter4\_3\_keras\_codesLast Checkpoint: 2018/11/24 (autosaved)

FileEditViewInsertCellKernelWidgetsHelp

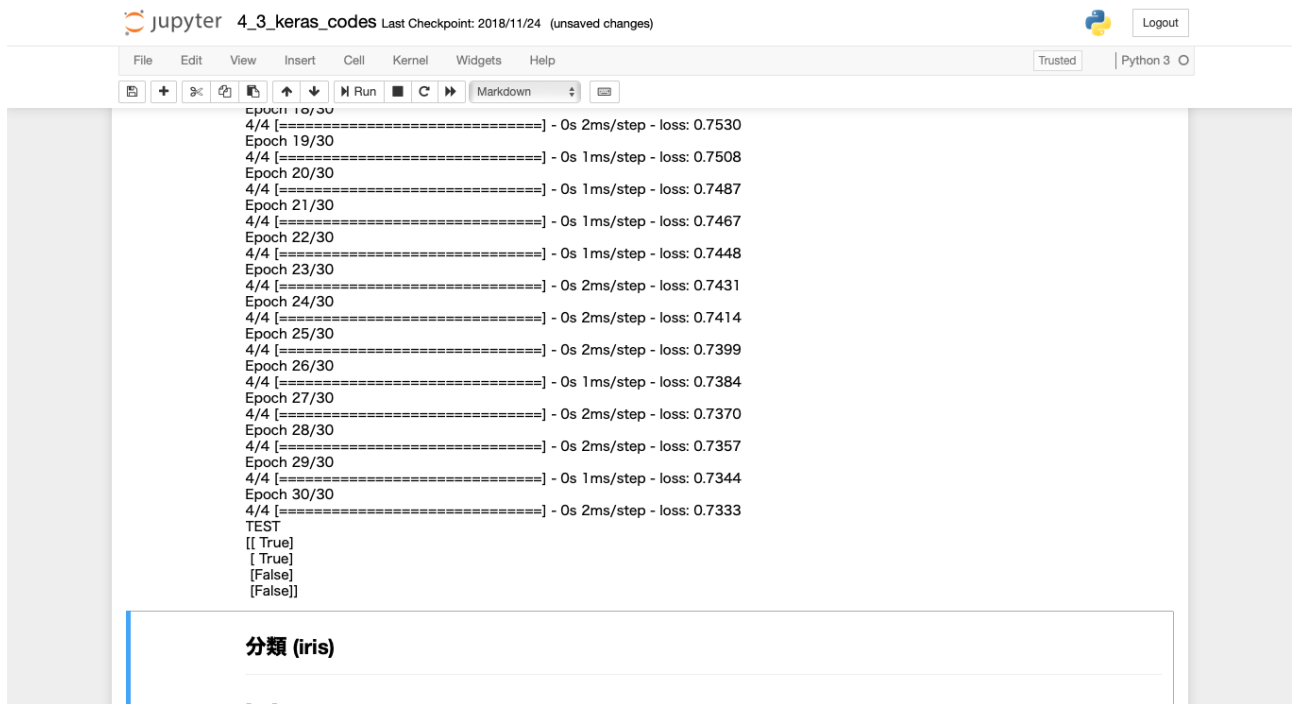
TrustedPython 3

Run

Code

```
4/4 [=====] - 0s 2ms/step - loss: 0.6195
Epoch 16/30
4/4 [=====] - 0s 2ms/step - loss: 0.6106
Epoch 17/30
4/4 [=====] - 0s 1ms/step - loss: 0.6023
Epoch 18/30
4/4 [=====] - 0s 2ms/step - loss: 0.5944
Epoch 19/30
4/4 [=====] - 0s 2ms/step - loss: 0.5870
Epoch 20/30
4/4 [=====] - 0s 2ms/step - loss: 0.5800
Epoch 21/30
4/4 [=====] - 0s 1ms/step - loss: 0.5733
Epoch 22/30
4/4 [=====] - 0s 879us/step - loss: 0.5671
Epoch 23/30
4/4 [=====] - 0s 833us/step - loss: 0.5612
Epoch 24/30
4/4 [=====] - 0s 850us/step - loss: 0.5556
Epoch 25/30
4/4 [=====] - 0s 1ms/step - loss: 0.5503
Epoch 26/30
4/4 [=====] - 0s 921us/step - loss: 0.5453
Epoch 27/30
4/4 [=====] - 0s 1ms/step - loss: 0.5406
Epoch 28/30
4/4 [=====] - 0s 933us/step - loss: 0.5361
Epoch 29/30
4/4 [=====] - 0s 1ms/step - loss: 0.5318
Epoch 30/30
4/4 [=====] - 0s 1ms/step - loss: 0.5277
TEST
[[ True]
 [False]
 [ True]
 [ True]]
```

## OR回路

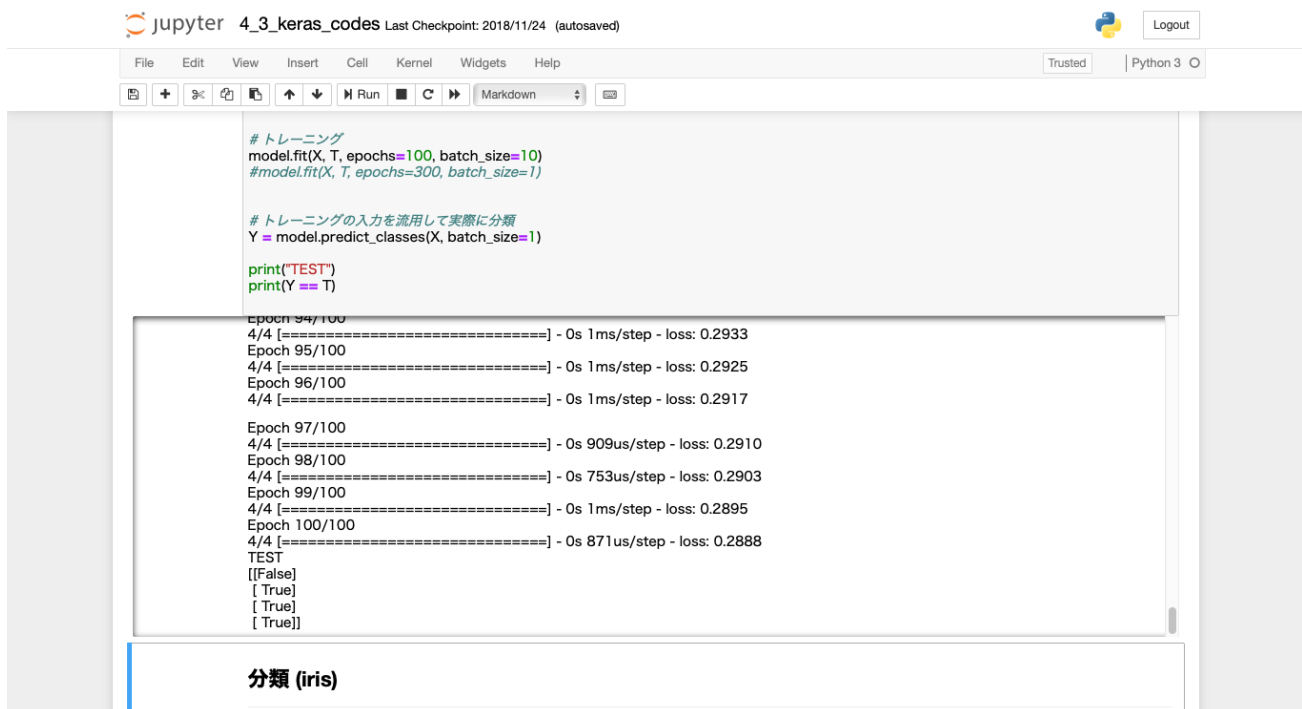


Jupyter 4\_3\_keras\_codes Last Checkpoint: 2018/11/24 (unsaved changes)

```
Epoch 19/30
4/4 [=====] - 0s 2ms/step - loss: 0.7530
Epoch 20/30
4/4 [=====] - 0s 1ms/step - loss: 0.7508
Epoch 21/30
4/4 [=====] - 0s 1ms/step - loss: 0.7487
Epoch 22/30
4/4 [=====] - 0s 1ms/step - loss: 0.7467
Epoch 23/30
4/4 [=====] - 0s 1ms/step - loss: 0.7448
Epoch 24/30
4/4 [=====] - 0s 2ms/step - loss: 0.7431
Epoch 25/30
4/4 [=====] - 0s 2ms/step - loss: 0.7414
Epoch 26/30
4/4 [=====] - 0s 2ms/step - loss: 0.7399
Epoch 27/30
4/4 [=====] - 0s 1ms/step - loss: 0.7384
Epoch 28/30
4/4 [=====] - 0s 2ms/step - loss: 0.7370
Epoch 29/30
4/4 [=====] - 0s 2ms/step - loss: 0.7357
Epoch 30/30
4/4 [=====] - 0s 1ms/step - loss: 0.7344
TEST
[[ True]
 [ True]
[False]
[False]]
```

分類 (iris)

## 4. OR回路でbatch\_size=10



Jupyter 4\_3\_keras\_codes Last Checkpoint: 2018/11/24 (autosaved)

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

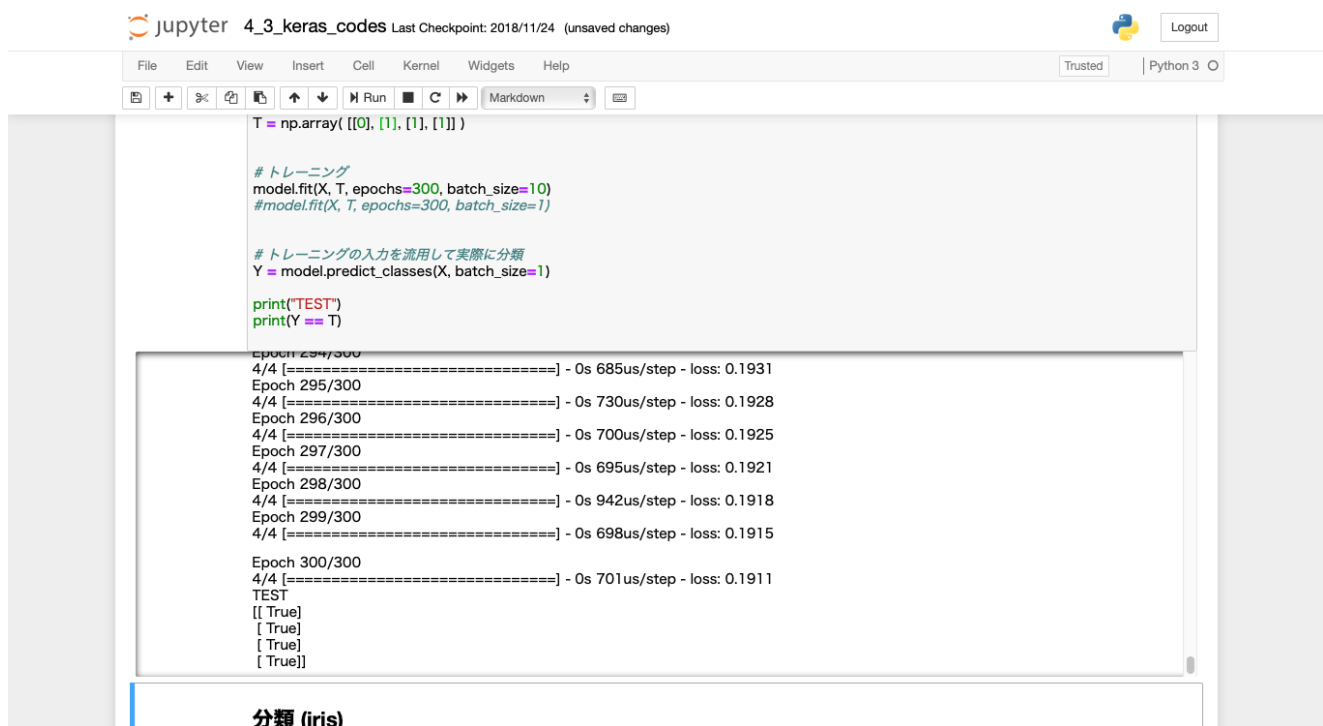
# トレーニングの入力を流用して実際に分類
Y = model.predict_classes(X, batch_size=1)

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

Epoch 94/100
4/4 [=====] - 0s 1ms/step - loss: 0.2933
Epoch 95/100
4/4 [=====] - 0s 1ms/step - loss: 0.2925
Epoch 96/100
4/4 [=====] - 0s 1ms/step - loss: 0.2917
Epoch 97/100
4/4 [=====] - 0s 909us/step - loss: 0.2910
Epoch 98/100
4/4 [=====] - 0s 753us/step - loss: 0.2903
Epoch 99/100
4/4 [=====] - 0s 1ms/step - loss: 0.2895
Epoch 100/100
4/4 [=====] - 0s 871us/step - loss: 0.2888
TEST
[[False]
 [ True]
 [ True]
 [ True]]
```

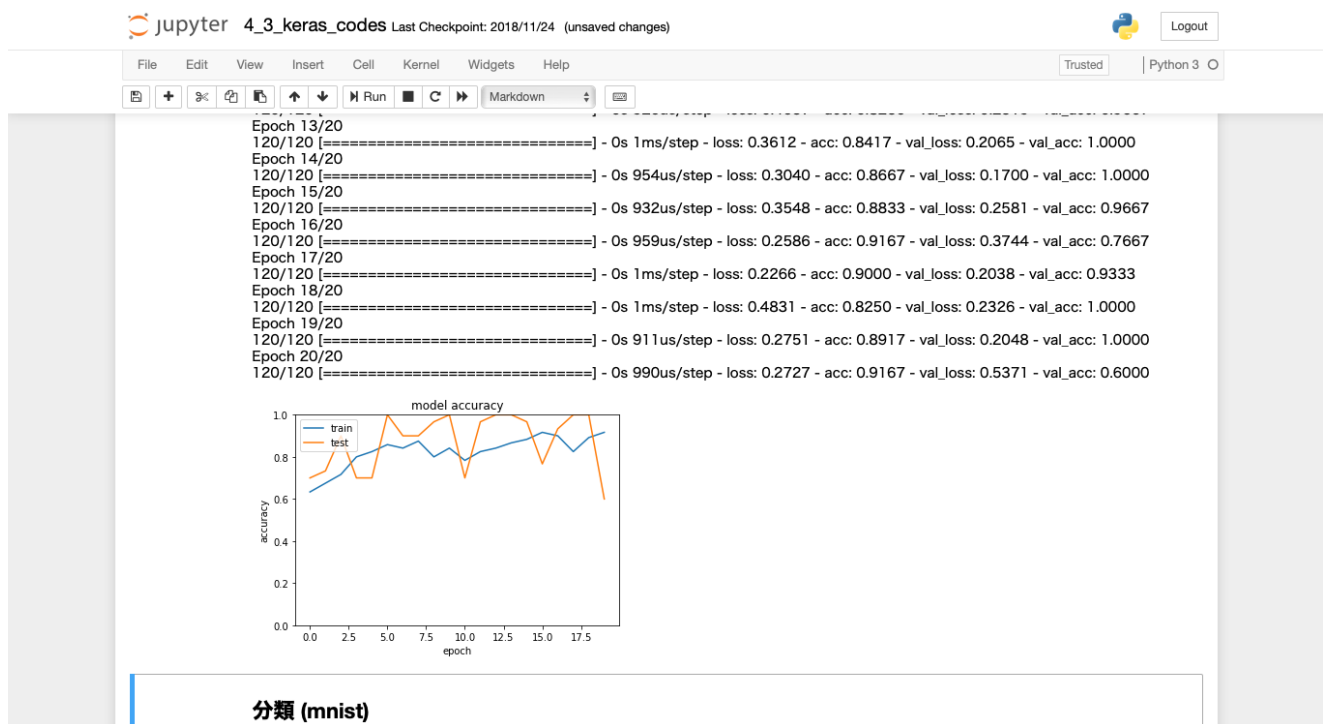
分類 (iris)

## 5.epoch=300 学習に時間がかかり、回答が変わる



バッチサイズは一般的には2の倍数(16,32,64,128...)にする。GPU的にはそのサイズが最適だから。  
OR回路は非線形なのでうまく学習できない。ニューロンを増やし、ReLUを使用するとOR回路でも正解率が上がる。

try : 分類(iris) sigmoid関数、learning\_rate=0.1 早い段階で精度が上がる。

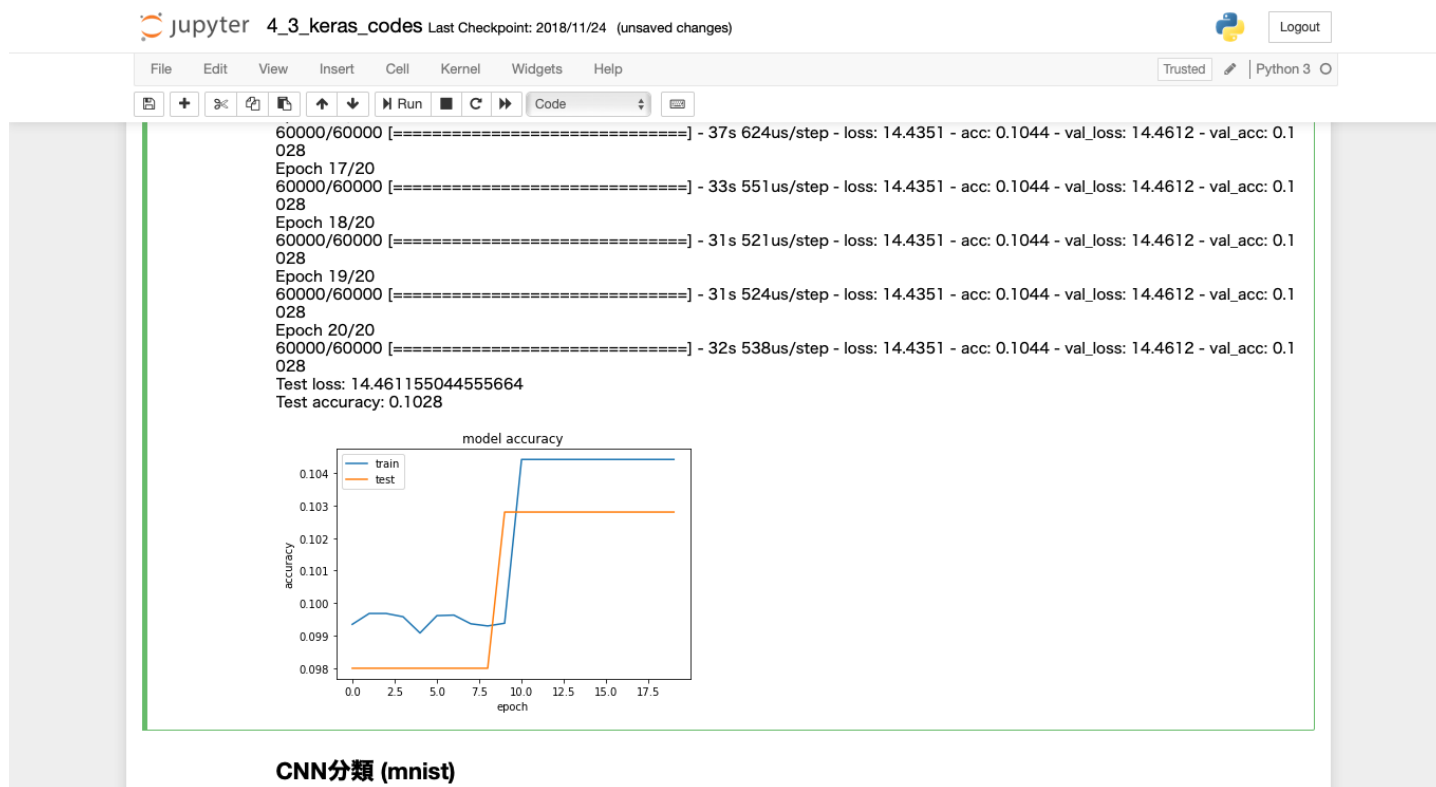




try: 分類(mnist) one\_hot\_label=False 実行エラー

誤差関数categorical\_crossentropyの時はone\_hot\_label=Trueでないと実行できない

誤差関数=sparse\_categorical\_crossentropy, one\_hot\_label=False, learning\_rate=0.5

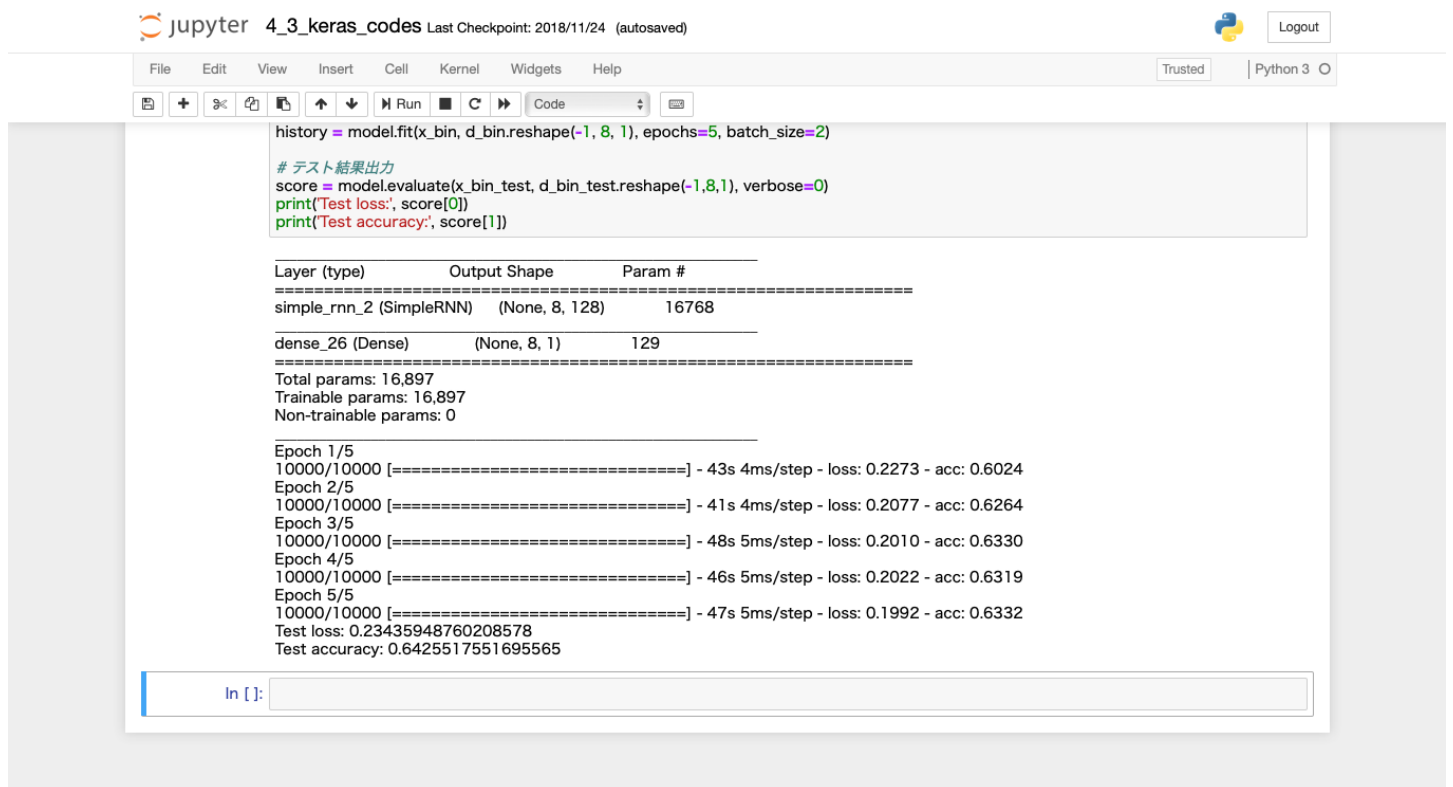


学習率を上げすぎるとうまく学習が進まなくなる。学習は早く終わる。

- CNN(mnist) Conv2D, MaxPooling2Dで簡単に実装できる。  
VGG16, ResNetもkerasで簡単に実装できる。
- cifar10 正規化～NNでは入力が0 or 1の小さなスケールにするために255で割る。  
BatchNormalizationをconvとReLUの間に入れると学習が早くなる。

try: RNN Sigmoidが8つあるので答えが8つ出力される。

unit=128, 活性化関数にtanh, optimizer=Adam, dropout=0.5, recurrent\_dropout=0.3で実行



```
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_2 (SimpleRNN)	(None, 8, 128)	16768
dense_26 (Dense)	(None, 8, 1)	129

Total params: 16,897  
Trainable params: 16,897  
Non-trainable params: 0

Epoch 1/5  
10000/10000 [=====] - 43s 4ms/step - loss: 0.2273 - acc: 0.6024  
Epoch 2/5  
10000/10000 [=====] - 41s 4ms/step - loss: 0.2077 - acc: 0.6264  
Epoch 3/5  
10000/10000 [=====] - 48s 5ms/step - loss: 0.2010 - acc: 0.6330  
Epoch 4/5  
10000/10000 [=====] - 46s 5ms/step - loss: 0.2022 - acc: 0.6319  
Epoch 5/5  
10000/10000 [=====] - 47s 5ms/step - loss: 0.1992 - acc: 0.6332  
Test loss: 0.23435948760208578  
Test accuracy: 0.6425517551695565

unitを増やしたので早い段階で精度が上がる。sigmoidでは精度はでないのでtanhにすると精度が上がる。SGDからAdamに変えたので精度が上がる。recurrenct\_dropoutを入れると精度は上がるが、スピードは落ちる。

KerasはGRU, LSTMも実装可能。GRUはSimpleRNNより遅いがLSTMよりは早い。

「Section2:強化学習」

- ・ 強化学習～長期的に報酬を最大化するように行動を選択できるエージェントを作ることが目標  
不完全な知識を元に行動しながらデータを収集し最適な行動を見つけてゆく。  
e.g: メールマガジンを誰に送るかを決めるアプリ、Googleのアルファ碁

- ・ 確認テスト エージェントの行動、方針の例：ゲームの場合  
行動：どうキャラクターを動かすか  
報酬：ゲームのポイント

- ・ 探索と利用のトレードオフ～行動しながら情報を集め最適な行動を見つけるトレードオフ。  
過去と同じ行動をとり続ける状態>探索が足りない

未知の行動をとり続ける状態＞利用が足りない

- ・ 概要：エージェント側 方策 $\pi$ （方策関数 $\pi(\text{state}, \text{action})$ ）、価値 $V$ (行動価値関数 $Q(s, a)$ )

環境側：状態 $S$

エージェントは行動を選択することで環境から報酬を得る。

強化学習は一連の行動を通じて報酬が最も多く得られるような方策(policy)を学習する。

教師ありなし学習はデータのパターンを見つけて予測するが、強化学習は優れた方策を探すのが目標

方策...ある状態 $s$ の時に行動 $a$ を取る確率

- ・ 歴史：冬の時代もあったが関数近似法とQ学習を組み合わせることで今の性能に。

Q学習～行動価値関数を行動することにより更新することにより学習をすすめる

関数近似法～価値関数や方策関数を関数近似する手法

- ・ 状態価値関数：状態 $s$ にいることの価値 $V(s)$
- ・ 行動価値関数：状態 $s$ において行動 $a$ をとる価値 $Q(s, a)$

状態価値関数

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^\pi(s')] \quad (1)$$

行動価値関数

$$Q^\pi(s, a) = \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^\pi(s')] \quad (2)$$

$$V^\pi(s) = \sum_a \pi(s, a) Q^\pi(s, a) \quad (3)$$

$$Q^\pi(s, a) = \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma \sum_{a'} \pi(s', a') Q^\pi(s', a')] \quad (4)$$

- ・ 報酬関数：観測した状態から取るべき行動を決定し、新たに遷移した結果として環境から受け取る値を求める関数 $R(s), R(s, a), R(s, a, s')$
- ・ 方策関数：状態 $s$ にの時に行動 $a$ をとる確率 $\pi(s, a)$ 。方策を学習するために累積報酬を最大化する。
- ・ 割引報酬～割引係数 $r(0 \leq r < 1)$ で割り引いた累積報酬。これにより現在の報酬より将来得られる報酬を重視するようになる。計算的に値が発散しない。時間が経つと環境が変化する可能性があるので時系列の全ての報酬を同じ重みで計算しない。
- ・ 平均報酬～行動をとった時に生まれる価値の全部の平均をとったもの。
- ・ 方策勾配法～方策関数をパラメータベクトルによって表現し、方策関数を直接最適化する方法

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

・ 例題 方策勾配定理の公式 答：a

・ 例題 DCGAN～CNNを使い敵対的学習を使い画像を生成する。

潜在空間でベクトル変換して画像を扱う。

Generatorが正規分布から画像を生成し、Discriminatorがその画像を判断する。

Generator, Discriminatorの学習が進んで理想的な性能を有するとGeneratorから生成される画像と訓練データの分布が一致するため、出力関数 $D(x)$ は入力データと訓練データの確率が1対1、つまり0.5になる。

・ 例題 GAN+CNN Generator = Fractionally Strided Convolution, 活性化関数ReLU

Discriminator = Strided Convolution, 活性化関数Leaky ReLU

Leaky ReLUの式：

$$f = x \ (x > 0), \ 0.01x \ (x \leq 0)$$