

深層学習 (day4)・要点まとめ

Section1 : Tensorflow の実装演習

1-1 Tensorflow による実装

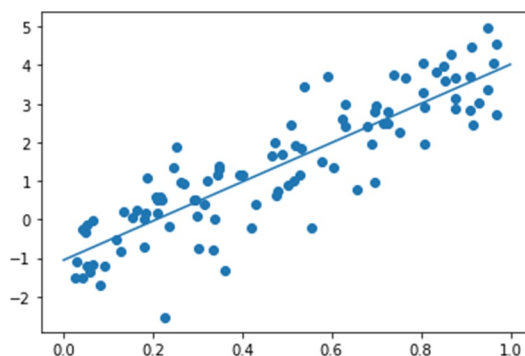
[実装上のポイント]:

- Session を起動するまでは、print 等の全ての命令は無効
- Placeholder は箱のようなもの 後で中身を変更できる

[線形回帰の実装の考察 (4_1_tensorflow_codes.ipynb)]:

- ノイズを大きくすると点群の広がりが大きくなり、線形回帰の式が目標の式とのズレが大きくなる。反対に、小さくすると点群の広がりが狭くなり、線形回帰の式が目標の式に近づく。
- 目標の式の係数やバイアスを変更すると、その直線の周囲に点群が点在することになる。

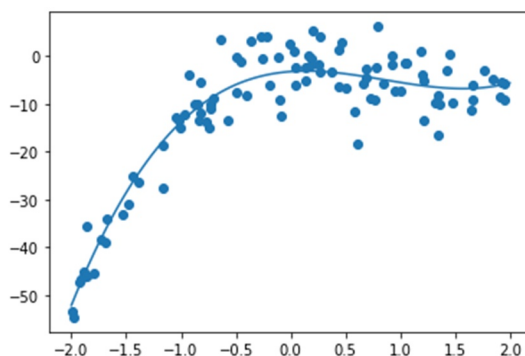
```
Generation: 280. 誤差 = 0.6589505
Generation: 290. 誤差 = 0.65875167
Generation: 300. 誤差 = 0.6586006
[5.0687222] [-1.051718]
```



[非線形回帰の実装の考察 (4_1_tensorflow_codes.ipynb)]:

- ノイズを大きくすると点群の広がりが大きくなり、非線形回帰の式の係数と目標の式の係数とのズレが大きくなる。反対に、小さくすると点群の広がりが狭くなり、両者の係数の値が近づく。
- 目標の式の係数を変更すると、その 3 次曲線の周囲に点群が点在することになる。

```
Generation: 9900. 誤差 = 21.931187
Generation: 10000. 誤差 = 21.877575
[[ 2.490546 ] [-6.403633 ] [ 1.6499753] [-3.2503297]]
```



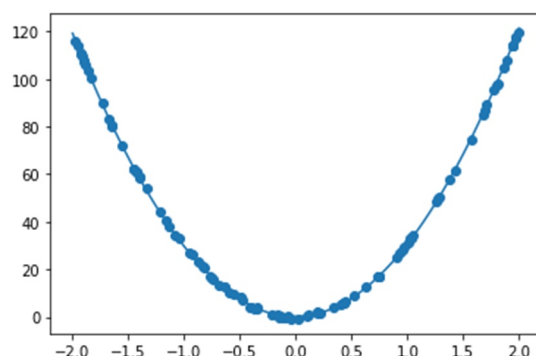
[演習問題の実装の考察 (4_1_tensorflow_codes.ipynb)]:

$y=30x^2+0.5x+0.2$ で回帰をするようにコードを修正

深層学習 (day4)・要点まとめ

- パラメータ数を 4 から 3 に減少するように配列の次元を変更。
- ノイズ 0.5 に対して、学習率を 0.1 に大きくすると、回帰の精度が極めて高くなった。エポック数は 3000 回で十分だった。

```
Generation: 2800. 誤差 = 0.2588178
Generation: 2900. 誤差 = 0.25881803
Generation: 3000. 誤差 = 0.2588178
[[30.0088 ] [ 0.539442 ] [ 0.17030227]]
```



[MNIST1 の実装の考察 (4_1_tensorflow_codes.ipynb)]:

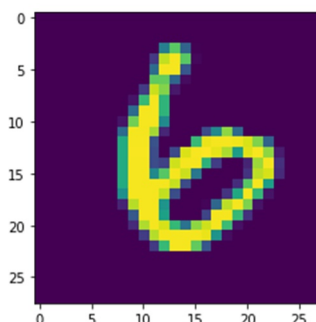
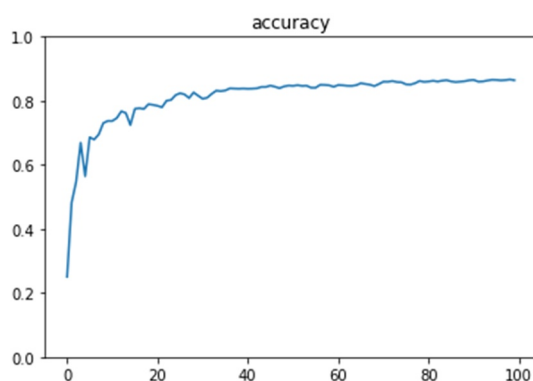
1) 分類 1 層 (中間層なし)

x: 入力値, d: 教師データ, W: 重み, b: バイアス をそれぞれ定義して実行。

```
[ True True True ... True False True]
Generation: 100. 正解率 = 0.8642
```

```
print("Target: ", d_batch[0])    # 最初の入力データに対するターゲットを表示
plt.imshow(x_batch[0].reshape(28, 28))    # 最初の入力データを表示
```

```
Target: [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]    # 最初のターゲット = "6"
```



2) 分類 3 層 (中間層 2 層)

□ 中間層 2 層のサイズによる違い:

オリジナル (600, 300) を (650, 450) に変更した方が、時間は要したものの精度は向上した (正解率 0.97 0.9724) 逆に、サイズを減らすと時間が短縮されるが、精度は下がった。

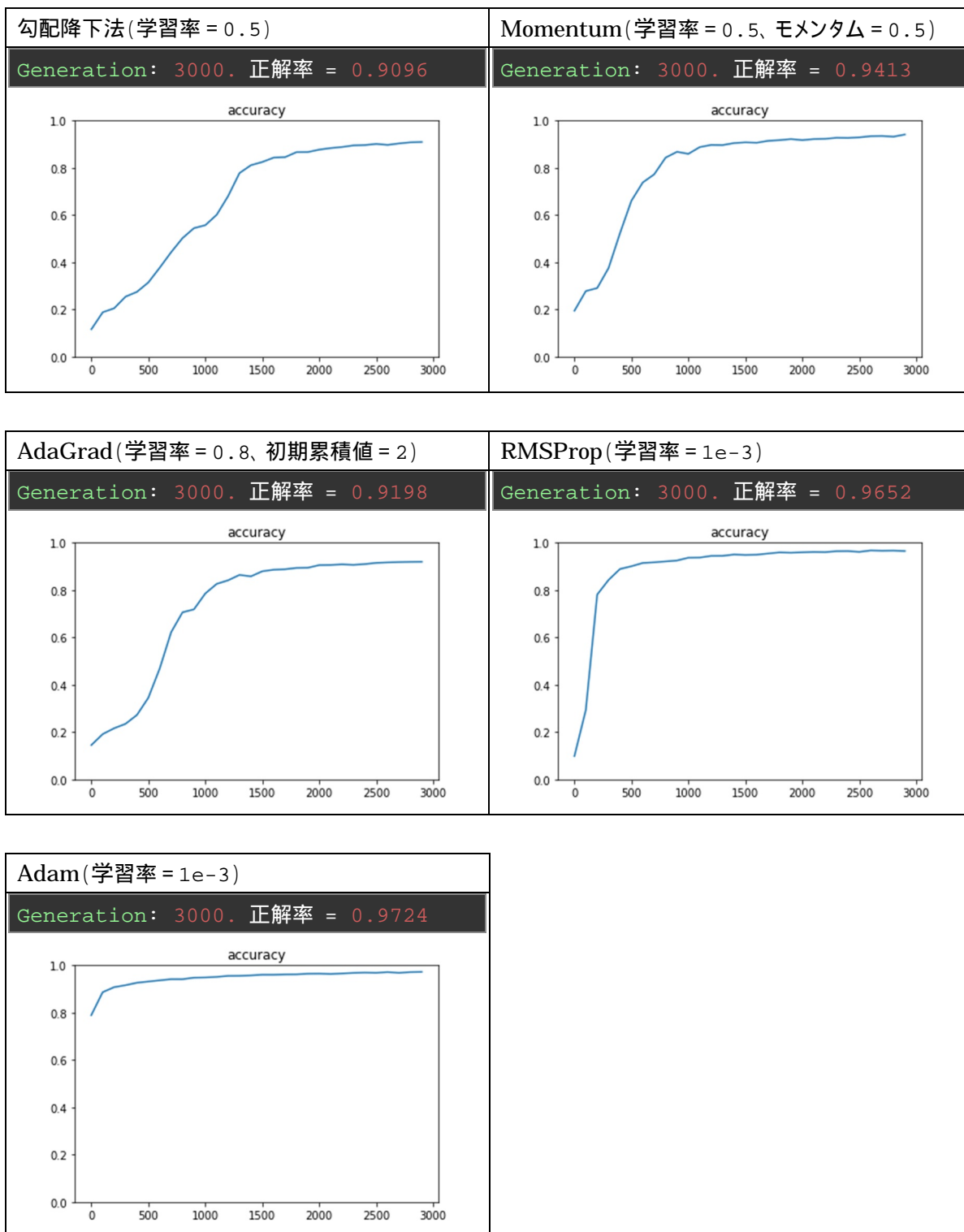
【考察】

1 層から 3 層に増やした方が、精度は大きく向上している。

中間層のサイズは、実行時間と精度のバランスで決める必要がある。

深層学習 (day4)・要点まとめ

□学習率最適化手法による違い(中間層のサイズ(650, 450)で実行):



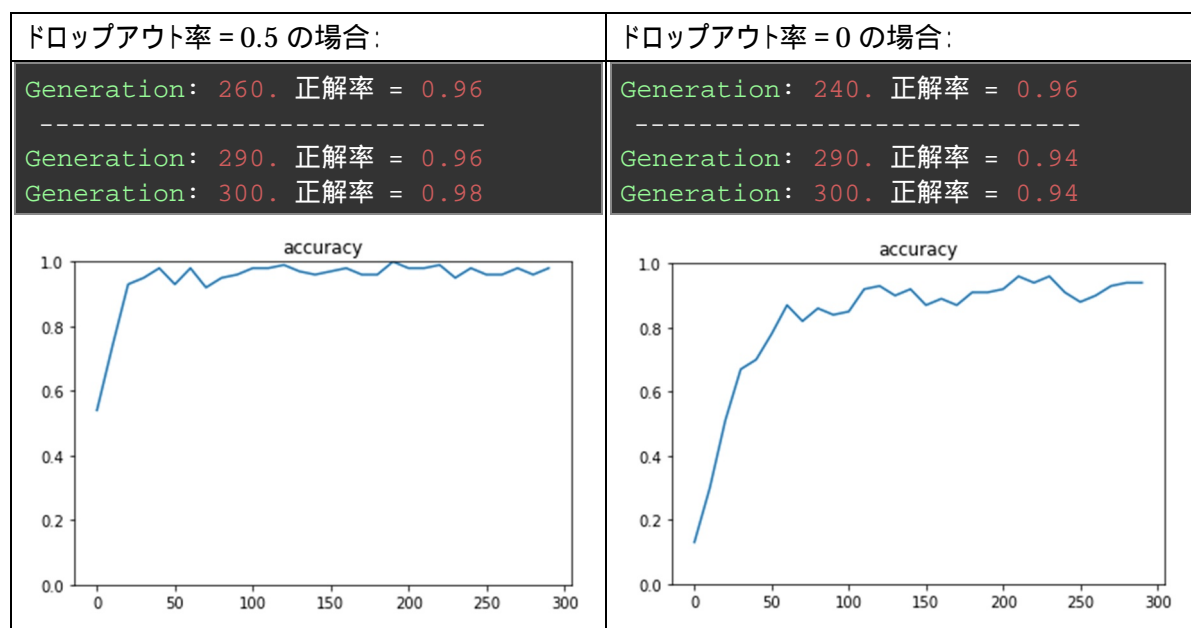
【考察】

- 精度は、Adam > RMSProp > モメンタム > AdaGrad > 勾配降下法 の順となった。ただ、学習率等のハイパーパラメータは、精度に敏感なので検討が必要なようである。
- やはり学習率最適化は、Adam か RMSProp のいずれかを使えば良いと思われる。

深層学習 (day4)・要点まとめ

3)分類 CNN

□ドロップアウト率の違い:



【考察】

- ドロップアウト率を 0 にした場合、0.5 の場合に比べてやや精度は劣るが、順調に学習は進み、エポック数 300 回では過学習は起こっていないようである。
- ただ、本コードでは訓練データとテストデータを分けておらず、精度はあくまで訓練データで行っているため、3 層分類の場合との精度比較はできない。もし、データ分割した場合は、ドロップアウトのあり / なしによる精度の違いは明らかに出てくるものと考えられる。

《確認テスト》: VGG、GoogLeNet、ResNet のそれぞれの特徴は？

- 1) VGG・・・convolution, convolution, max_pooling という単純なネットワークの積み重ねの構成である。一方で、パラメータ数は GoogLeNet や ResNet に比べて多い点の特徴。
- 2) GoogLeNet・・・inception module を使っていて、中でも 1×1 フィルターの畳み込みによる次元削減や様々なフィルターサイズを使うことでスパースな演算となる点の特徴。
- 3) ResNet・・・skip connection、identity module を使うことによって残差接続を行い、それによって深い層であっても勾配消失を解決している点の特徴。

1-2 Keras による実装

【線形回帰の実装の考察(4_3_keras_codes.ipynb)】:

```
Generation: 280. 誤差 = 0.6589505
```

```
Model: "sequential_1"
```

```
Layer (type) Output Shape Param #
```

```
=====
```

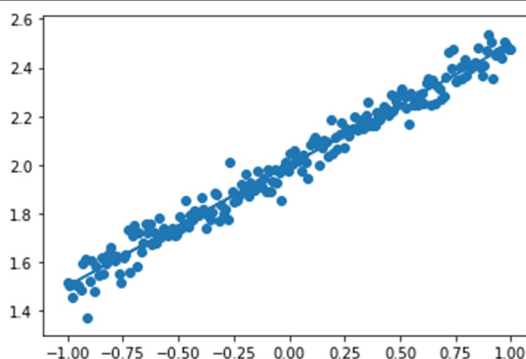
```
dense_1 (Dense) (None, 1) 2
```

```
=====
```

深層学習 (day4)・要点まとめ

```
Total params: 2 Trainable params: 2 Non-trainable params: 0
```

```
-----  
Generation: 990. 誤差 = 0.0021703965  
Generation: 1000. 誤差 = 0.0021703478  
w: [[0.4959172]]  
b: [1.9990368]
```



[考察] Tensorflow に比べて、Placeholder や Variables の定義が必要なく、コード全体がシンプルである。特に、モデルの作成が非常にシンプルである。

[単純パーセプトロンの実装の考察 (4_3_keras_codes.ipynb)]:

- OR 回路への入力信号を単純パーセプトロンで分類させる。乱数シードが固定では上手く分類できていたが、シードを変えるとエポック数 30 回、バッチサイズ 1 で実行しても上手く分類できない。エポック数を 100 回に増やすと 2 値交差エントロピー誤差が小さくなり正解の分類ができた。
- AND 回路に変えて実行(エポック数 100 回)しても正解の分類ができた。ただし、XOR 回路では正解の分類ができない。
- OR 回路で、バッチサイズを 1 から 10 に増やすとエポック数 100 回では分類できない。エポック数を 300 回に増やすと、上手く分類できた。

```
np.random.seed(1)  
  
# OR 回路  
X = np.array( [[0,0], [0,1], [1,0], [1,1]] )  
T = np.array( [[0], [1], [1], [1]] )  
  
model.fit(X, T, epochs=30, batch_size=1)      # OR 回路(エポック=30)==> ×  
Epoch 30/30  
4/4 [=====] - 0s 3ms/step - loss: 0.2961  
TEST  
[[False]  
 [ True]  
 [ True]  
 [ True]]  
  
model.fit(X, T, epochs=100, batch_size=1)     # OR 回路(エポック=100)==> ○  
Epoch 100/100  
4/4 [=====] - 0s 2ms/step - loss: 0.1715  
TEST  
[[ True]  
 [ True]  
 [ True]  
 [ True]]
```

深層学習 (day4)・要点まとめ

```
# AND 回路 ==> ○
X = np.array( [[0,0], [0,1], [1,0], [1,1]] )
T = np.array( [[0], [0], [0], [1]] )

Epoch 100/100
4/4 [=====] - 0s 2ms/step - loss: 0.2527
TEST
[[ True]
 [ True]
 [ True]
 [ True]]

# XOR 回路 ==> ×
X = np.array( [[0,0], [0,1], [1,0], [1,1]] )
T = np.array( [[0], [1], [1], [0]] )

Epoch 100/100
4/4 [=====] - 0s 2ms/step - loss: 0.7196
TEST
[[ True]
 [False]
 [ True]
 [False]]
```

[考察] 乱数シードを変えると上手く分類できないのは、初期値が変わったためと思われる。OR 回路や AND 回路では単純パーセプトロンによる線形で上手く分類できるが、XOR 回路では分類できないのだろう。XOR 回路は、非線形でないと上手く分類できないようである。以下のように 1 層追加し、活性化関数に ReLU を設定し非線形にすると、上手く分類できた。

```
model.add(Dense(input_dim=2, units=5, activation='relu'))
```

```
# OR 回路

model.fit(X, T, epochs=100, batch_size=10)
# OR 回路(エポック=100、バッチサイズ=10)==> ×
Epoch 100/100
4/4 [=====] - 0s 750us/step - loss: 0.3063
TEST
[[False]
 [ True]
 [ True]
 [ True]]

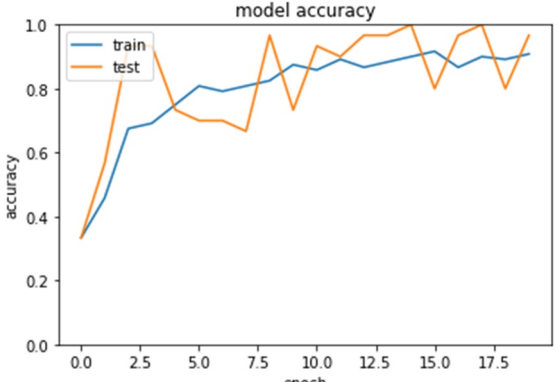
model.fit(X, T, epochs=300, batch_size=10)
# OR 回路(エポック=300、バッチサイズ=10)==> ○
Epoch 300/300
4/4 [=====] - 0s 1ms/step - loss: 0.2007
TEST
[[ True]
 [ True]
 [ True]
 [ True]]
```

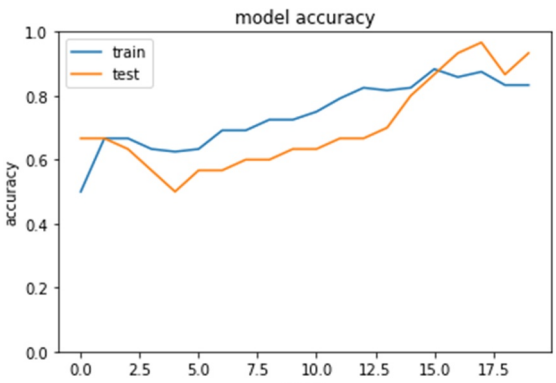
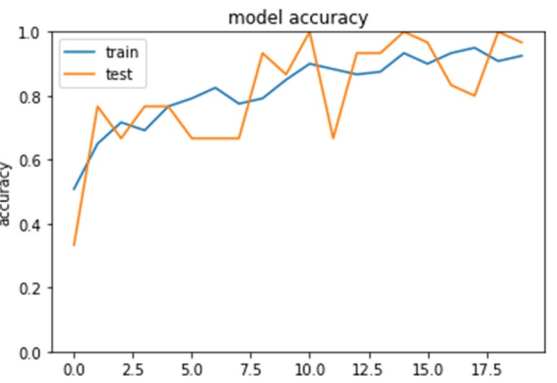
[考察] バッチサイズ 10 に増やしたことによって誤差関数の値が大きくなり、エポック数を増やしないと学習し切れなかったためと思われる。

深層学習 (day4)・要点まとめ

[分類 IRIS の実装の考察 (4_3_keras_codes.ipynb)]:

- 入力層 4 ノード、中間層 12 ノード、出力層 3 ノードの 3 層のモデル(下記参照)で IRIS データを分類(3 種類から 1 つを推定)。中間層の活性化関数が ReLU 関数の場合、シグモイド関数に変えた場合、更に、SGD の学習率を 0.01 から 0.1 にした場合で学習・予測を実行した。
- 活性化関数を変えた場合、ReLU 関数の方が学習速度・精度共に高い結果となった。
- シグモイド関数で学習率を 0.1 に大きくした場合、学習速度・精度共に高い結果となった。

モデル	活性化関数:ReLU、学習率 = 0.01
<pre>Epoch 20/20 Model: "sequential_4" Layer (type) Output Shape Param # ===== dense_7 (Dense) (None, 12) 60 activation_7 (Activation) (None, 12) 0 dense_8 (Dense) (None, 3) 39 activation_8 (Activation) (None, 3) 0 ===== Total params: 99 Trainable params: 99 Non-trainable params: 0</pre>	<pre>Epoch 20/20 120/120 [==== 0s 367us/step =====] - loss: 0.3203 - acc: 0.9083 - val_loss: 0.2671 - val_acc: 0.9667</pre> 

活性化関数:シグモイド、学習率 = 0.01	活性化関数:シグモイド、学習率 = 0.1
<pre>Epoch 20/20 120/120 [==== 0s 275us/step =====] - loss: 0.7158 - acc: 0.8333 - val_loss: 0.7737 - val_acc: 0.9333</pre> 	<pre>Epoch 20/20 120/120 [==== 0s 333us/step =====] - loss: 0.2685 - acc: 0.9250 - val_loss: 0.2438 - val_acc: 0.9667</pre> 

[考察] 活性化関数を変えた場合、ReLU 関数の方が学習速度・精度共に高い結果となったが、過学習には陥っていないため、エポック回数を増やせば、両方とも精度がより向上するものと考えられる。

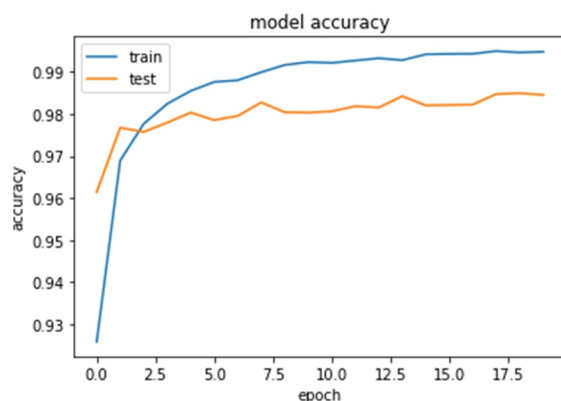
深層学習 (day4)・要点まとめ

[分類(mnist)の実装の考察 (4_3_keras_codes.ipynb)]:

1) One hot encoding で学習・評価:

- モデルは、入力層 784 - 中間層 512 - 中間層 512 - 出力層 10 で、0 から 9 までの手書き文字を One hot encoding で学習・評価を実施。結果、汎化性能が 98.4%となった。

```
Epoch 20/20 60000/60000 [=====] - 20s 333us/step  
- loss: 0.0161 - accuracy: 0.9948 - val_loss: 0.0731 - val_accuracy: 0.9845  
Test loss: 0.07312285845366405 Test accuracy: 0.984499990940094
```



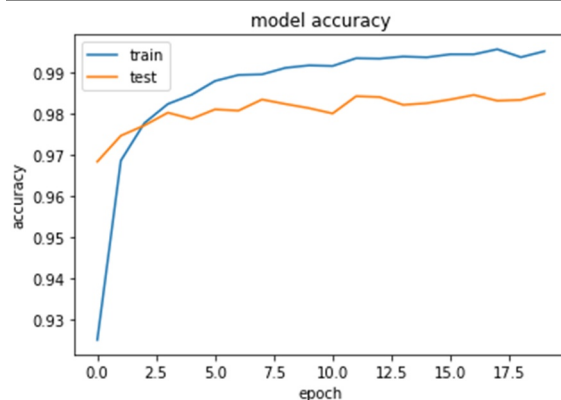
[考察]

- 中間層 2 層でも汎化性能が 98.4%となったが、エポック数を増やすともう少し精度は向上すると考えられる。

2) カテゴリ分類で学習・評価:

- カテゴリ分類(ラベルが 0 から 9 までの整数)で学習・評価を実施。結果、汎化性能が 98.5%となった。

```
Epoch 20/20 60000/60000 [=====] - 19s 318us/step  
- loss: 0.0146 - accuracy: 0.9951 - val_loss: 0.0749 - val_accuracy: 0.9848  
Test loss: 0.07494800291256251 Test accuracy: 0.9847999811172485
```



[考察]

- ★ 中間層 2 層でも汎化性能が 98.4%となったが、エポック数を増やすともう少し精度は向上すると考えられる。
- ★ ラベルが one-hot エンコーディングで表現されている場合は、損失関数は `categorical_crossentropy` を使用し、ラベルが整数の場合は、損失関数は `sparse_categorical_crossentropy` を使用する。

3) Adam のパラメータ変更:

- Adam のパラメータをデフォルトの値 (`lr=0.001`, `beta_1=0.9`, `beta_2=0.999`) から変更したが、汎化性能が 98.5%を超えることができなかった。

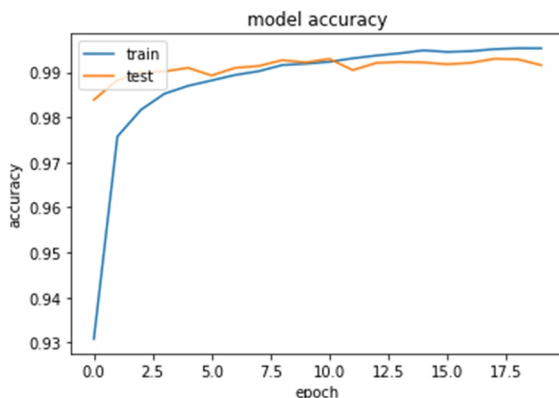
[考察] 複数のハイパーパラメータを手動で変えて最適地を求めることはやはり難しい。やがり、グリッドサーチやベイズ最適化等の手法を用いることが必要だろう。

[CNN 分類(mnist)の実装の考察 (4_3_keras_codes.ipynb)]:

- 畳み込み + 畳み込み + 最大プーリング + 全結合層のモデルで、手書き文字 (mnist) の分類を、オリジナルの条件のまま実行。結果、エポック数が 20 回で汎化性能は 99.2%であった。

深層学習 (day4)・要点まとめ

```
Epoch 18/20
60000/60000 [=====] - 310s 5ms/step
- loss: 0.0141 - accuracy: 0.9952 - val_loss: 0.0303 - val_accuracy: 0.9930
Epoch 19/20
60000/60000 [=====] - 286s 5ms/step
- loss: 0.0140 - accuracy: 0.9954 - val_loss: 0.0305 - val_accuracy: 0.9929
Epoch 20/20
60000/60000 [=====] - 296s 5ms/step
- loss: 0.0133 - accuracy: 0.9953 - val_loss: 0.0336 - val_accuracy: 0.9916
```



[考察]

★ 過学習の状態なのでこれ以上精度は高くないが、画像分類における CNN の効果が確認できた。

[cifar10 の実装の考察 (4_3_keras_codes.ipynb)]:

- 畳み込み + 畳み込み + 最大プーリングを2つ重ねたモデルで自然画像の分類を、オリジナルの条件のまま実行。結果、エポック数が 20 回で汎化性能は 0.83 であった。

```
Epoch 18/20
50000/50000 [=====] - 492s 10ms/step - loss: 0.5190 - accuracy: 0.8164
Epoch 19/20
50000/50000 [=====] - 464s 9ms/step - loss: 0.5072 - accuracy: 0.8222
Epoch 20/20
50000/50000 [=====] - 474s 9ms/step - loss: 0.4942 - accuracy: 0.8253
10000/10000 [=====] - 32s 3ms/step
[0.6526181121826172, 0.7886000275611877]
```

[考察] エポック数を増やせばもう少し精度は向上すると考えられるが、自然画像の分類においては、比較的シンプルな CNN モデルではこれが限界かもしれない。

[Simple RNN(2 進数足し算の予測)の実装の考察 (4_3_keras_codes.ipynb)]:

1) 出力ノード数の変更:

- 出力ノード数を 16 から 128 に増やすと、実行速度は遅くなるが汎化性能は 100%となる。(出力ノード数が 16 の場合でも 100%になる)

出力ノード数 = 16	出力ノード数 = 128
Epoch 1/5 - loss: 0.0771 - accuracy: 0.9178	Epoch 1/5 - loss: 0.0710 - accuracy: 0.9286
Epoch 2/5 - loss: 0.0029 - accuracy: 1.0000	Epoch 2/5 - loss: 0.0017 - accuracy: 1.0000
Epoch 3/5 - loss: 9.5192e-04 - accuracy: 1.0000	Epoch 3/5 - loss: 6.3249e-04 - accuracy: 1.0000
Epoch 4/5 - loss: 5.3812e-04 - accuracy: 1.0000	Epoch 4/5 - loss: 3.7655e-04 - accuracy: 1.0000

深層学習 (day4)・要点まとめ

```
Epoch 5/5
- loss: 3.6653e-04 - accuracy: 1.0000
Test loss: 0.000311029474329153
Test accuracy: 1.0
```

```
Epoch 5/5
- loss: 2.6339e-04 - accuracy: 1.0000
Test loss: 0.00022737540966361806
Test accuracy: 1.0
```

[考察] 本問題のようなシンプルな場合は精度面ではあまり差がないが、複雑な問題ではその差は出てくるものと考えられる。

2) 出力活性化関数を変更:

- ReLU からシグモイドに変わると精度は低下した。tanh では精度は再度向上した。

```
[sigmoid]:
Epoch 5/5
10000/10000 [=====] - 34s 3ms/step - loss: 0.1780 - accuracy: 0.7588
Test loss: 0.11756535312353653
Test accuracy: 0.8681368231773376

[tanh]:
Epoch 5/5
10000/10000 [=====] - 32s 3ms/step - loss: 2.1175e-04 - accuracy: 1.0000
Test loss: 0.00018430571535910556
Test accuracy: 1.0
```

[考察] RNN では時間軸に渡って勾配が小さくなるので、活性化関数は tanh の方が合っているようである。ただし、ReLU と比べてどうかまでは評価できない。

3) 最適化法を SGD から Adam に変更:

- 活性化関数をシグモイドに戻して、勾配降下の最適化法を SGD から Adam に変えて実行。結果、精度 100%まで向上できた。

```
[Adam(sigmoid)]:
Epoch 1/5
10000/10000 [=====] - 37s 4ms/step - loss: 0.2483 - accuracy: 0.5378
Epoch 2/5
10000/10000 [=====] - 36s 4ms/step - loss: 0.2351 - accuracy: 0.6500
Epoch 3/5
10000/10000 [=====] - 36s 4ms/step - loss: 0.1304 - accuracy: 0.8938
Epoch 4/5
10000/10000 [=====] - 36s 4ms/step - loss: 0.0166 - accuracy: 1.0000
Epoch 5/5
10000/10000 [=====] - 36s 4ms/step - loss: 0.0013 - accuracy: 1.0000
Test loss: 0.0003068407746577224
Test accuracy: 1.0
```

[考察] Adam にする効果は大きいようである。

4) Dropout の設定:

- 入力 Dropout を 50%にして実行したが、設定前より汎化性能が低下した。
- 再帰 Dropout を 30%にして実行した。汎化性能は、設定前よりも低下しているが、入力 Dropout よりも精度は高くなった。

```
[入力 Dropout=0.5]:
Epoch 5/5
```

深層学習 (day4)・要点まとめ

```
10000/10000 [=====] - 58s 6ms/step - loss: 0.2346 - accuracy: 0.5985
Test loss: 0.219319970384337
Test accuracy: 0.6883438229560852

[再帰 Dropout=0.3]:
Epoch 1/5
10000/10000 [=====] - 73s 7ms/step - loss: 0.2498 - accuracy: 0.5178
Epoch 2/5
10000/10000 [=====] - 72s 7ms/step - loss: 0.2424 - accuracy: 0.5883
Epoch 3/5
10000/10000 [=====] - 73s 7ms/step - loss: 0.2042 - accuracy: 0.7074
Epoch 4/5
10000/10000 [=====] - 72s 7ms/step - loss: 0.1616 - accuracy: 0.7955
Epoch 5/5
10000/10000 [=====] - 74s 7ms/step - loss: 0.1291 - accuracy: 0.8358
Test loss: 0.091214589664329
Test accuracy: 0.9148289561271667
```

[考察] RNN では、中間層が時間軸に渡って伝搬するため、入力側よりもドロップアウトによる影響が大きくなることが原因と考えられる。

5) Unroll の設定:

- 再帰 Dropout を 30% のまま、Unroll = True にして実行。実行速度が約 1/3 に短縮されると共に、汎化性能も設定前よりも改善している。

```
[Unroll=True(再帰 Dropout=0.3)]:
Epoch 1/5
10000/10000 [=====] - 25s 3ms/step - loss: 0.2499 - accuracy: 0.5166
Epoch 2/5
10000/10000 [=====] - 25s 3ms/step - loss: 0.2406 - accuracy: 0.6024
Epoch 3/5
10000/10000 [=====] - 25s 2ms/step - loss: 0.1929 - accuracy: 0.7179
Epoch 4/5
10000/10000 [=====] - 25s 2ms/step - loss: 0.1302 - accuracy: 0.8324
Epoch 5/5
10000/10000 [=====] - 25s 2ms/step - loss: 0.0821 - accuracy: 0.9091
Test loss: 0.0420767759734934
Test accuracy: 0.9562081098556519
```

[考察]、Unroll = True にすると、ネットワークはメモリ上に展開され、RNN がスピードアップされている。精度が向上したのは、設定前のループ処理において精度面で割り切りがされているためと考えられる。

unroll: 真理値 (デフォルトは False) . True なら、ネットワークは展開され、そうでなければシンボリックループが使われる。展開はよりメモリ集中傾向になるが、RNN をスピードアップできる。展開は短い系列にのみ適している。

深層学習 (day4)・要点まとめ

Section2 : 強化学習

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が目標
行動の結果として与えられる報酬を基に、行動を決定する原理を改善する仕組み

2-2. 強化学習の応用例

<マーケティングの場合> :

- 環境 = 会社の販売促進部
- エージェント = プロフィールと購買履歴に基づいて、キャンペーンメールを送る顧客を決めるソフトウェア
- 方策 = キャンペーンメールを送る顧客を決める
- 価値 = キャンペーンのコストという負の報酬と、キャンペーンで生み出されると推測される売上という正の報酬を受ける

2-3. 探索と利用のトレードオフ

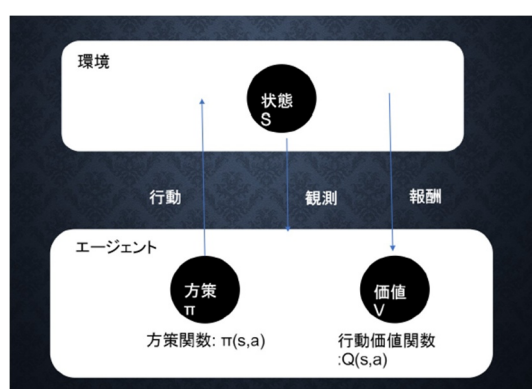
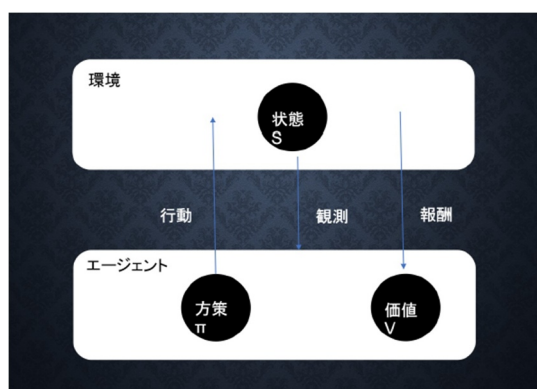
環境に関する事前知識がなく、最適な行動を予測し決定することはできない(どのような顧客にキャンペーンメールを送付すると、どのように行動するかはわからない)

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る

探索が足りない状態と利用が足りない状態とのトレードオフ:

- 探索が足りない状態 = 過去のデータや経験だけでベストと思われる行動だけでは、他に存在するもっとベストな行動を見つけられない
- 利用が足りない状態 = 未知の行動のみを常に取り続けければ、過去の経験が生かせない

2-4. 強化学習のイメージ



- エージェントが環境の中で、自身が得る収益を最大化するために行動を選択し、それによって変化する状態の中で、最終的に最適な方策を獲得する。

2-5. 強化学習の差分

- 教師あり、教師なし学習: データに含まれるパターンを見つけ出すこと、そのデータから予測することが目標 強化学習: 優れた方策を見つけることが目標
- 計算速度の進展により大規模な状態を持つ場合の強化学習が可能、関数近似法と Q 学習を組

深層学習 (day4)・要点まとめ

み合わせる手法の登場 強化学習の発展

Q 学習 = 行動価値関数を、行動する毎に更新することにより学習を進める方法

関数近似法 = 価値関数や方策関数を関数近似する手法

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がある。

状態価値関数: ある状態の価値に注目する場合

行動価値関数: 状態と価値を組み合わせた価値に注目する場合 $Q(s,a)$

2-7. 方策関数

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探るかの確率を与える関数

$$\pi(s,a)$$

2-8. 方策勾配法

方策をモデル化して最適化する手法: 方策勾配法

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

J : 方策の良さ... 定義が必要

定義方法: ・平均報酬 ・割引報酬和

この定義に対応して、行動価値関数 $Q(s,a)$ を定義し、方策勾配定理が成立する

$$\nabla_{\theta} J(\theta) = E_{\pi_{\theta}} \left[\left(\nabla_{\theta} \log \pi_{\theta}(a|s) Q^{\pi}(s|a) \right) \right]$$

【考察】

ある状態において、状態価値関数の勾配が最大となる方策となるようにパラメータ θ を更新する。