@gucchi0403 2019年01月03日に更新

ResNetをいろんな機械学習ライブラリで実装してみた~TensorFlow編~

Python 機械学習 TensorFlow ResNet

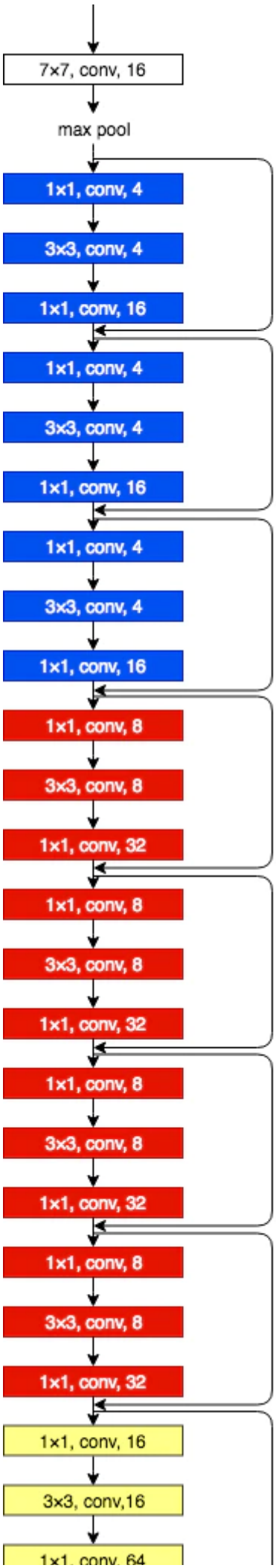
⚠ この記事は最終更新日から1年以上が経過しています。

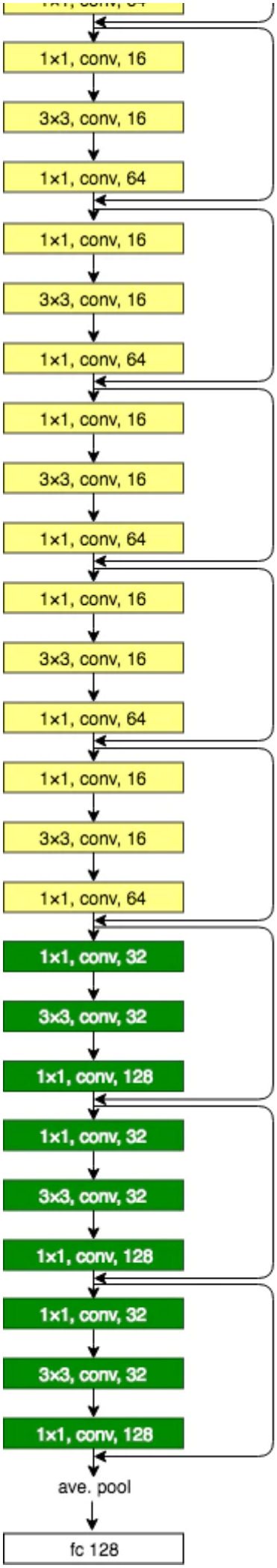
機械学習にはライブラリがたくさんあって、どのライブラリを使えばいいかわかんない。
なので、それぞれのライブラリの計算速度とコード数をResNetを例に測ってみます。
今回はTensorFlow編です。他はKeras, Chainer, PyTorchでやってみる予定。
今回のコードはnotebook形式でGitHubにあげてます。
https://github.com/RyosukeSakaguchi/ResNet/blob/master/TensorFolw.ipynb

今回実装したResNetについて

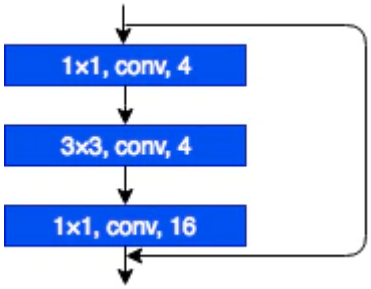
論文:http://arxiv.org/abs/1512.03385
分かりやすい日本語の解説:https://deepage.net/deep_learning/2016/11/30/resnet.html

今回はResNet-50を実装した。全体のネットワークの図は以下の通りである。

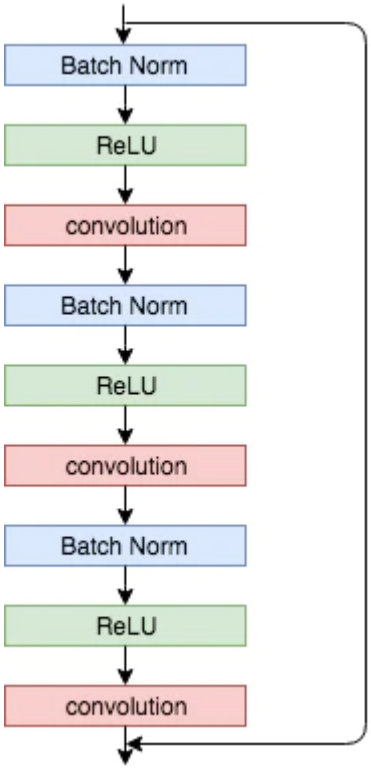

7×7, conv, 16
max pool
1×1, conv, 4
3×3, conv, 4
1×1, conv, 16
1×1, conv, 4
3×3, conv, 4
1×1, conv, 16
1×1, conv, 4
3×3, conv, 4
1×1, conv, 16
1×1, conv, 8
3×3, conv, 8
1×1, conv, 32
1×1, conv, 8
3×3, conv, 8
1×1, conv, 32
1×1, conv, 8
3×3, conv, 8
1×1, conv, 32
1×1, conv, 8
3×3, conv, 8
1×1, conv, 32
1×1, conv, 16
3×3, conv,16
1×1, conv, 64


1×1, conv, 16
3×3, conv, 16
1×1, conv, 64
1×1, conv, 16
3×3, conv, 16
1×1, conv, 64
1×1, conv, 16
3×3, conv, 16
1×1, conv, 64
1×1, conv, 16
3×3, conv, 16
1×1, conv, 64
1×1, conv, 16
3×3, conv, 16
1×1, conv, 64
1×1, conv, 32
3×3, conv, 32
1×1, conv, 128
1×1, conv, 32
3×3, conv, 32
1×1, conv, 128
1×1, conv, 32
3×3, conv, 32
1×1, conv, 128
ave. pool
fc 128

Bottleneckアーキテクチャ16個と最初の1つの畳み込み層と最後の全結合1層で合計50層です。
Bottleneckアーキテクチャでは、1つのResidual Blockが3つの畳み込み層を含み、以下の構造になっています。



Bottleneckアーキテクチャの他には、2つの畳み込み層を含むPlainアーキテクチャがあります。
また、活性化関数とBatch Normalizationを畳み込み層の前に持ってくるPre Activationを用いました。1つのResidual BlockでのPre Activationは以下のような構成です。



Pre Activationの他には、Batch Normalizationを後の方に持ってくるPost Activationがありますが、Pre Activationの方が一般的に精度がいいみたいです。
OptimizerはSGD+Momentumを使用しました。

環境

GPUが無料で使える、との事でGoogle Colaboratoryで行いました。セットアップは↓のURLを参考に。
https://qiita.com/tomo_makes/items/f70fe48c428d3a61e131

教師データについて

今回は、ResNetを使って皆さんご存知の手書き文字MNISTのクラス分けをします。
学習は全55000枚の画像で、バッチサイズは128で、エポック数は10にしました。
テストは全10000枚の画像で、バッチサイズは128で、エポック数は10にしました。

本題

ResNetのコードはhttps://github.com/xuyuwei/resnet-tf を参考にしました。(このコードは物体認識のベンチマークであるCIFAR-10専用のコードになってますんで、今回はMNIST用にカスタマイズしました。)

まずはMNISTをダウンロードします。tensorflowのtutorialにMNISTデータがあるので、そこからダウンロードします。以下のコードを実行すると、 `./MNIST_data` 配下にデータがぶち込まれます。(`one_hot=True` とするとラベルがone-hotベクトルで取得できる)
tensorflowのtutorialのMNISTの詳細は以下のページを参照。
http://tensorflow.classcat.com/2016/03/09/tensorflow-cc-mnist-for-ml-beginners/

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

また、以下のようにResNetの最初の畳み込み層に入力できるように学習データの入力データを整形する関数を用意する。

```
def data_train(batch_size):
  # MNISTの全学習データからbatch_size(int)個のデータをランダムに取り出し、
  # 入力データをx_trainに入れ、y_trainにラベルを入れる。
```

```
  # 入力データは784次元のベクトル、ラベルは10次元のone-hotベクトル
  x_train, y_train = mnist.train.next_batch(batch_size)

  # 畳み込み層に入力できるように入力データを整形。
  # 出来上がったx_train_dataは形状(batch_size, 28, 28, 1)のNumPy配列
  # 28×28の行列で、白黒なのでチャンネル数は1。
  x_train_data = []
  for data in x_train:
    x_train_data.append(np.reshape(data, (28, 28,1)))
  x_train_data = np.array(x_train_data)

  return x_train_data, y_train
```

テストデータも同様の関数を用意。

```
def data_test(batch_size):
# MNISTの全テストデータからbatch_size(int)個のデータをランダムに取り出し、
  # 入力データをx_testに入れ、y_testにラベルを入れる。
  # 入力データは784次元のベクトル、ラベルは10次元のone-hotベクトル
  x_test, y_test = mnist.test.next_batch(batch_size)

  # 畳み込み層に入力できるように入力データを整形。
  # 出来上がったx_test_dataは形状(batch_size, 28, 28, 1)のNumPy配列
  # 28×28の行列で、白黒なのでチャンネル数は1。
  x_test_data = []
  for data in x_test:
    x_test_data.append(np.reshape(data, (28, 28,1)))
  x_test_data = np.array(x_test_data)

  return x_test_data, y_test
```

では、実際にResNetを作成していく。
以下はResNetの各層のコードである。residual_block関数は上で説明した通り、Pre Activationで作成している。

```
import numpy as np
import tensorflow as tf

def weight_variable(shape, name=None):
    # 各層で使用する重み行列を返す関数
    # 標準偏差が0.1の切断正規分布からshapeで指定された形のテンソルを生成し、initialに代入
    initial = tf.truncated_normal(shape, stddev=0.1)
```

```
    initial = tf.truncated_normal(shape, stddev=0.1)

    # 初期のテンソルがinitialの変数tf.Variableを返す
    return tf.Variable(initial, name=name)

def softmax_layer(inpt, shape):
    # shapeで指定される形の重みをfc_wに代入
    fc_w = weight_variable(shape)

    # shape[1]で指定される形の重みをfc_bに代入
    #初期値はゼロベクトル
    fc_b = tf.Variable(tf.zeros([shape[1]]))

    # 全結合後、ソフトマックスを計算
    fc_h = tf.nn.softmax(tf.matmul(inpt, fc_w) + fc_b)

    return fc_h

def conv_layer(inpt, filter_shape, stride):
    # 入力データのチャンネル数をinpt_channelsに代入
    inpt_channels =  inpt.get_shape().as_list()[3]

    # Batch Normalization
    # チャンネル毎に平均meanと分散varを計算
    mean, var = tf.nn.moments(inpt, axes=[0,1,2])
    # Batch Normalizationに使用する学習パラメータbetaとgammaを準備
    # betaの初期値はゼロベクトル
    beta = tf.Variable(tf.zeros([inpt_channels]), name="beta")
    gamma = weight_variable([inpt_channels], name="gamma")
    # Batch Normalization実施
    batch_norm = tf.nn.batch_norm_with_global_normalization(
        inpt, mean, var, beta, gamma, 0.001,
        scale_after_normalization=True)

    # 活性化関数としてReLU関数使用
    out_relu = tf.nn.relu(batch_norm)

    # 畳み込み層
    # filter_shapeで指定される形の重みをfilter_に代入
    filter_ = weight_variable(filter_shape)
    # 畳み込み層の出力をoutに代入
    out = tf.nn.conv2d(out_relu, filter=filter_, strides=[1, stride, stride, 1], padding="

    return out

def residual_block(inpt, output_depth, stride=1, projection=False):
```

```
def residual_block(inpt, output_depth, stride=1, projection=False):
    # 入力データのチャンネル数をinput_depthに代入
    input_depth = inpt.get_shape().as_list()[3]

    # Batch Normalization + Relu +畳み込みを3セット
    conv1 = conv_layer(inpt, [1, 1, input_depth, int(output_depth/4)], stride)
    conv2 = conv_layer(conv1, [3, 3, int(output_depth/4), int(output_depth/4)], stride)
    conv3 = conv_layer(conv2, [1, 1, int(output_depth/4), output_depth], stride)

    # 入力と出力のチャンネル数が異なる場合は以下の2つの方法でチャンネル数を揃える
    if input_depth != output_depth:
        if projection:
            # Option B: Projection shortcut
            input_layer = conv_layer(inpt, [1, 1, input_depth, output_depth], 2)
        else:
            # Option A: Zero-padding
            # 足りない部分を0でパディング
            input_layer = tf.pad(inpt, [[0,0], [0,0], [0,0], [0, output_depth - input_dept
    else:
        input_layer = inpt

    # conv3に入力を足す
    res = conv3 + input_layer

    return res
```

次に上の各層の関数を用いて、以下resnet関数を定義する。

```
import tensorflow as tf

def resnet(inpt):

    layers = []

    # Residual Blockに入る前に1つ畳み込み層とmax poolingを通す
    with tf.variable_scope('conv1'):
        conv1 = conv_layer(inpt, [7, 7, 1, 16], 1)
        max_pooling = tf.nn.max_pool(conv1, [1, 3, 3, 1], [1, 1, 1, 1], padding="SAME")
        layers.append(conv1)
        layers.append(max_pooling)

    # residual blockの総数は3個
    # 出力のshapeは[batch_size, 28, 28, 16]
```

```
with tf.variable_scope('conv2'):
    conv2_1 = residual_block(layers[-1], 16)
    conv2_2 = residual_block(conv2_1, 16)
    conv2_3 = residual_block(conv2_2, 16)
    layers.append(conv2_1)
    layers.append(conv2_2)
    layers.append(conv2_3)

assert conv2_3.get_shape().as_list()[1:] == [28, 28, 16]

# residual blockの総数は4個
# 出力のshapeは[batch_size, 28, 28, 32]
with tf.variable_scope('conv3'):
    conv3_1 = residual_block(layers[-1], 32, stride=1)
    conv3_2 = residual_block(conv3_1, 32)
    conv3_3 = residual_block(conv3_2, 32)
    conv3_4 = residual_block(conv3_3, 32)
    layers.append(conv3_1)
    layers.append(conv3_2)
    layers.append(conv3_3)
    layers.append(conv3_4)

assert conv3_4.get_shape().as_list()[1:] == [28, 28, 32]

# residual blockの総数は6個
# 出力のshapeは[batch_size, 28, 28, 64]
with tf.variable_scope('conv4'):
    conv4_1 = residual_block(layers[-1], 64, stride=1)
    conv4_2 = residual_block(conv4_1, 64)
    conv4_3 = residual_block(conv4_2, 64)
    conv4_4 = residual_block(conv4_3, 64)
    conv4_5 = residual_block(conv4_4, 64)
    conv4_6 = residual_block(conv4_5, 64)
    layers.append(conv4_1)
    layers.append(conv4_2)
    layers.append(conv4_3)
    layers.append(conv4_4)
    layers.append(conv4_5)
    layers.append(conv4_6)

assert conv4_6.get_shape().as_list()[1:] == [28, 28, 64]

# residual blockの総数は3個
# 出力のshapeは[batch_size, 28, 28, 128]
with tf.variable_scope('conv5'):
```

```
        conv5_1 = residual_block(layers[-1], 128, stride=1)
        conv5_2 = residual_block(conv5_1, 128)
        conv5_3 = residual_block(conv5_2, 128)
        layers.append(conv5_1)
        layers.append(conv5_2)
        layers.append(conv5_3)

    assert conv5_3.get_shape().as_list()[1:] == [28, 28, 128]

    with tf.variable_scope('fc'):
        # batch_sizeとチャンネル数毎に平均をとる
        global_pool = tf.reduce_mean(layers[-1], [1, 2])

        assert global_pool.get_shape().as_list()[1:] == [128]

        # 全結合+ソフトマックス
        out = softmax_layer(global_pool, [128, 10])
        layers.append(out)

    return layers[-1]
```

最後にResNet-50で訓練とテストを行う、main関数を作成。

```
def main():
    batch_size = 128

    X = tf.placeholder("float", [batch_size, 28, 28, 1])
    Y = tf.placeholder("float", [batch_size, 10])
    learning_rate = tf.placeholder("float", [])

    # ResNet
    net = resnet(X)

    cross_entropy = -tf.reduce_sum(Y*tf.log(net))
    opt = tf.train.MomentumOptimizer(learning_rate, 0.9)
    train_op = opt.minimize(cross_entropy)

    sess = tf.Session()
    sess.run(tf.initialize_all_variables())

    correct_prediction = tf.equal(tf.argmax(net, 1), tf.argmax(Y, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
```

```
    saver = tf.train.Saver()

    train_step_num = int(55000 / batch_size)
    test_step_num = int(10000 / batch_size)
    epoch_num = 10

    # 学習
    for j in range (epoch_num):
        for i in range(train_step_num):
            x_train, y_train = data_train(batch_size)
            feed_dict={
                X: x_train,
                Y: y_train,
                learning_rate: 0.001}
            sess.run([train_op], feed_dict=feed_dict)

    # テスト
    accs = []
    for j in range (epoch_num):
        for i in range(test_step_num):
            x_test, y_test = data_test(batch_size)
            acc = sess.run([accuracy],feed_dict={
                X: x_test,
                Y: y_test
            })
            accuracy_summary = tf.summary.scalar("accuracy", accuracy)
            accs.append(acc[0])

    sess.close()

    return sum(accs)/len(accs)
```

このmain関数を実行すると訓練とテストが始まる。

```
import time
start = time.time()
acc = main()
print('精度 : ' + str(acc))
print('時間 : ' + str(time.time() - start) + 's')
```

実行結果は以下です。

出力

```
精度 : 0.9865084134615385
時間 : 1807.990844488144s
```

結果

TensorFlowを用いた場合、ResNet-50は1808秒(30分8秒)でした。これを他のライブラリの計算時間と比較します。
また、コメントを除いたコード行数148行でした。これも他のライブラリのコード行数と比較します。
精度は0.9865ですか。もっと高くできるはずなので、精度向上はまたの機会に行いますね。
次はKerasでResNet-50を実装し、計算時間とコード行数を算出します。

諒輔 坂口 @gucchi0403

フォロー

コメント

この記事にコメントはありません。

あなたもコメントしてみませんか :)

ユーザー登録

すでにアカウントを持っている方はログイン