Reparametrization Trick for Batch Normalization

Proof of unaltered Accuracy

14/04/2018 Alberto IBARRONDO | Melek ONEN

This notebook serves as proof of the Reparametrization Trick used to absorb a Batch Normalization (BN) layer by the immediately previous Fully Connected (FC) or Convolutional (Conv) layer. To prove it, we will:

1. Setup and CNN implementation using Tensorflow.
2. Train a Convolutional Neural Network (CNN) containing BN layers from scratch, using MNIST training dataset.
3. Evaluate accuracy of such CNN using the MNIST test dataset.
4. Extract all the parameters from the trained CNN.
5. Define a CNN with the same architecture except for the BN layers, that will be deleted.
6. Perform the reparametrization trick to alter the weights and biases in Conv and FC layers of the original CNN and load them in the CNN without BN.
7. Evaluate accuracy of reparametrized CNN using the MNIST test dataset, comparing it to the original. For an easy reproducibility of our proof, we include all the functions and CNNet class in `CNNet.py`. They can be easily imported using:

```
from CNNet import *
```

We have also saved the models `tinyCNN` (with BN) and `NoBN_tinyCNN` (without BN and reparametrized) inside the `Models/` folder.

Setup & Dataset

We will use Tensorflow to implement the CNNs, and NumPy to perform the reparametrization trick.

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

As stated before, we will use the MNIST dataset, normalized to [0,1].

In [3]:
```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

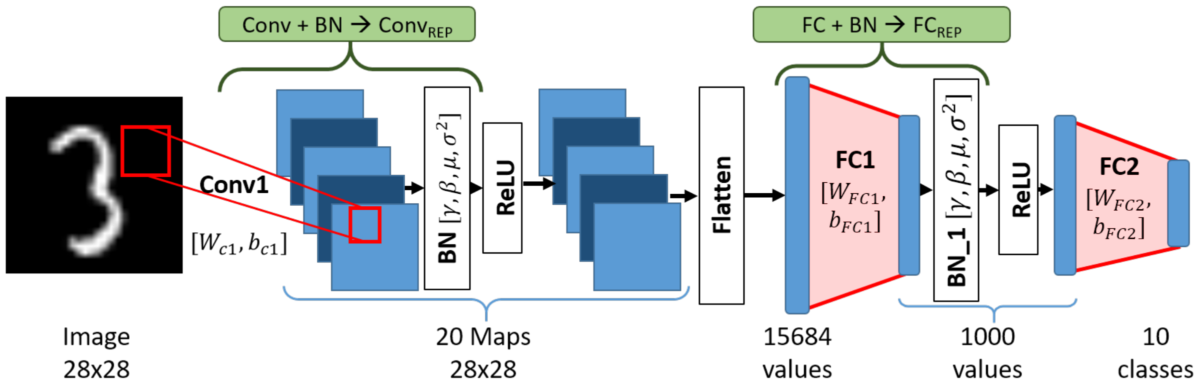
CNN implementation in Tensorflow

CNNet implements a simple CNN with only one Conv layer and two FC layers, with this architecture:

```
    28x28 => Conv1 -> BN -> ReLU -> Flat -> FC1 -> BN_1 -> ReLU -> FC2
=> [0-9]
```

The details of the network architecture are:

- **Conv1**: 20 Kernels/filters of 5x5, 20 biases. {1x28x28 -> 20x28x28}
- **BN**: 20 betas, gammas, moving averages and variances. {20x28x28 -> 20x28x28}
- **ReLU**: no parameters. {20x28x28 -> 20x28x28}
- **Flat**: {20x28x28 -> 15684}
- **FC1**: 15684*1000 weights, 1000 biases. {15684 -> 1000}
- **BN_1**: 1000 betas, gammas, moving averages and variances. {1000 -> 1000}
- **ReLU**:no parameters. {1000 -> 1000}
- **FC1**: 1000*10 weights, 10 biases. {1000 -> 10}



With this architecture we can use the reparametrization trick in **Conv1 + BN** and in **FC1 + BN_1**.

Our implemented python class **CNNet** takes charge of everything. It includes the following methods:

- *init*: setup the whole tf graph and session.
- *del*: clean the tf session before deleting the object.
- *CNN_var_init*: initializing weights and biases as tf variables.
- *preproc*: simple preprocessing for the input data (mean substraction).
- *CNN_feedforward*: define the feedforward operations of the network.
- *batchNorm*: wrapper of Batch Normalization in Tensorflow.
- *save & restore*: handle different models/instances.
- *train*: use MNIST train dataset to train the model.
- *benchmark*: use one of the MNIST dataset parts (train/validation/test) to evaluate accuracy.
- *predict*: perform inference on a single image.

ReLU will be our chosen activation function. Additionally, we initialize weights randomly and biases to a small constant. For all of the definitions we use the standard Tensorflow API.

In [5]:
```
from CNNet import CNNet
```

Training a CNN with BN from stratch

To speed up training, we will perform two phases of 5 steps each, one with learning rate 0.01 and the second with learning rate 0.002 for finer tuning.

In [9]:
```
cnn = CNNet('tinyCNN', lr=0.01)
```

In [10]:
```
cnn.train(epochs=5)
```

```
tinyCNN
* START TRAIN {l_r: 0.0100; epochs: 5; batch: 128, TrAcc: 8.5 %, ValAcc: 8.2 %}
  [13.3] Epoch: 01 | Loss=0.177882215 | ValAcc=96.340
  [26.6] Epoch: 02 | Loss=0.051323252 | ValAcc=96.820
  [39.8] Epoch: 03 | Loss=0.032282230 | ValAcc=97.980
  [52.8] Epoch: 04 | Loss=0.020916438 | ValAcc=98.120
  [65.9] Epoch: 05 | Loss=0.023993346 | ValAcc=97.940
* END TRAIN in 65.9 seconds => TestAcc: 97.790
```

In [11]:
```
cnn.save('Models/tinyCNN')
```

```
INFO: TF Model saved in file: Models/tinyCNN
```

In [12]:
```
cnn = CNNet('tinyCNN', lr=0.002, loadInstance='Models/tinyCNN')
```

```
INFO:tensorflow:Restoring parameters from Models/tinyCNN
```

In [13]:
```
cnn.train(epochs=5)
```

```
tinyCNN
* START TRAIN {l_r: 0.0020; epochs: 5; batch: 128, TrAcc: 99.1 %, ValAcc: 97.9 %}
  [13.2] Epoch: 01 | Loss=0.006767424 | ValAcc=98.860
  [26.2] Epoch: 02 | Loss=0.001558920 | ValAcc=98.900
  [39.3] Epoch: 03 | Loss=0.000715273 | ValAcc=98.960
  [52.3] Epoch: 04 | Loss=0.000532333 | ValAcc=98.980
  [65.4] Epoch: 05 | Loss=0.000429326 | ValAcc=98.940
* END TRAIN in 65.4 seconds => TestAcc: 99.020
```

In [14]:
```
cnn.save('Models/tinyCNN')
```

```
INFO: TF Model saved in file: Models/tinyCNN
```

Evaluating CNN with BN

In [15]:
```
cnn.benchmark('TEST')
```

Out[15]:
```
0.9902000010013581
```

As a result of the brief training, our trained CNN with BN layers yields a **test accuracy of 99.02%**.

Extracting parameters from trained CNN

For the sake of completeness, we will be verbose and show all the parameters being extracted from the current model (obtained from the default tf graph):

- Trainable variables

In [16]:
```
trainVariables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES)
trainVariables
```

Out[16]:
```
[<tf.Variable 'Model/W_c1:0' shape=(5, 5, 1, 20) dtype=float32_ref>,
 <tf.Variable 'Model/b_c1:0' shape=(20,) dtype=float32_ref>,
 <tf.Variable 'Model/W_fc1:0' shape=(15680, 1000) dtype=float32_ref>,
 <tf.Variable 'Model/b_fc1:0' shape=(1000,) dtype=float32_ref>,
 <tf.Variable 'Model/W_fc2:0' shape=(1000, 10) dtype=float32_ref>,
 <tf.Variable 'Model/b_fc2:0' shape=(10,) dtype=float32_ref>,
 <tf.Variable 'BatchNorm/beta:0' shape=(20,) dtype=float32_ref>,
 <tf.Variable 'BatchNorm/gamma:0' shape=(20,) dtype=float32_ref>,
 <tf.Variable 'BatchNorm_1/beta:0' shape=(1000,) dtype=float32_ref>,
 <tf.Variable 'BatchNorm_1/gamma:0' shape=(1000,) dtype=float32_ref>]
```

- Moving means and variances from BN layers

In [17]:
```
modelVariables = tf.get_collection(tf.GraphKeys.MODEL_VARIABLES)[2:4] + \
tf.get_collection(tf.GraphKeys.MODEL_VARIABLES)[6:8]
modelVariables
```

Out[17]:
```
[<tf.Variable 'BatchNorm/moving_mean:0' shape=(20,) dtype=float32_ref>,
 <tf.Variable 'BatchNorm/moving_variance:0' shape=(20,) dtype=float32_ref>,
 <tf.Variable 'BatchNorm_1/moving_mean:0' shape=(1000,) dtype=float32_ref>,
 <tf.Variable 'BatchNorm_1/moving_variance:0' shape=(1000,) dtype=float32_ref>]
```

We will save them all inside a dictionary due to its ease of access:

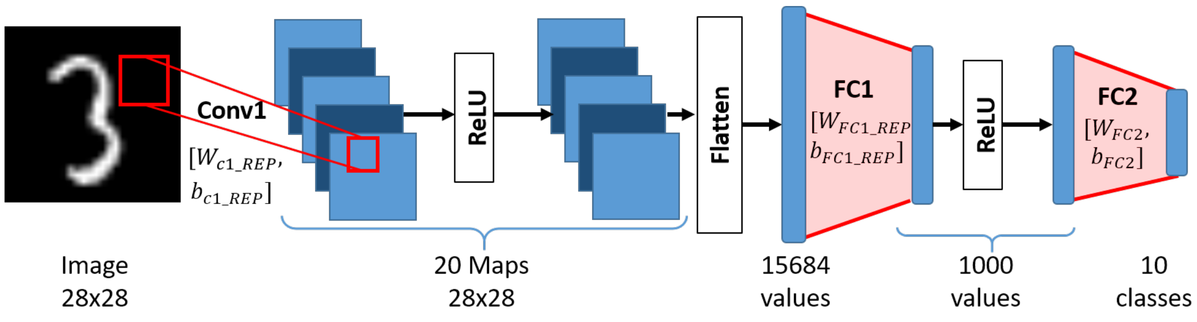
In [19]:
```
allVariables = {}
for var in trainVariables:
    allVariables[var.name]=var.eval(session=cnn.sess)
for var in modelVariables:
    allVariables[var.name]=var.eval(session=cnn.sess)
```

In [20]:
```
allVariables.keys()
```

Out[20]:
```
dict_keys(['Model/W_c1:0', 'Model/b_c1:0', 'Model/W_fc1:0', 'Model/b_fc1:0', 'Model/W
_fc2:0', 'Model/b_fc2:0', 'BatchNorm/beta:0', 'BatchNorm/gamma:0', 'BatchNorm_1/beta:
0', 'BatchNorm_1/gamma:0', 'BatchNorm/moving_mean:0', 'BatchNorm/moving_variance:0',
'BatchNorm_1/moving_mean:0', 'BatchNorm_1/moving_variance:0'])
```

Defining a CNN without BN

This CNN has exactly the same architecture as the previous one, save for the BN layers, that have been erased:



In [21]:
```
NoBNcnn = CNNet('NoBN_tinyCNN', lr=0.01, flag_bNorm=False)
```

Since we haven't trained this network, its initial test accuracy (13.6%) is based on random initialization of the weights, and thus it is similar to random picking a class for each image (10% chance of success).

In [22]:
```
NoBNcnn.benchmark('TEST')
```
Out[22]:
```
0.1362000025808811
```

If we assign the parameters of the trained tinyCNN (without assigning any BN parameters, since there are none in this model) we obtain some kind of improvement in accuracy, but still far from optimal:

In [ ]:
```
NoBNcnn.sess.run(NoBNcnn.W_c1.assign(allVariables['Model/W_c1:0']))
NoBNcnn.sess.run(NoBNcnn.b_c1.assign(allVariables['Model/b_c1:0']))
NoBNcnn.sess.run(NoBNcnn.W_fc1.assign(allVariables['Model/W_fc1:0']))
NoBNcnn.sess.run(NoBNcnn.b_fc1.assign(allVariables['Model/b_fc1:0']))
NoBNcnn.sess.run(NoBNcnn.W_fc2.assign(allVariables['Model/W_fc2:0']))
NoBNcnn.sess.run(NoBNcnn.b_fc2.assign(allVariables['Model/b_fc2:0']))
```

In [26]:
```
NoBNcnn.benchmark('TEST')
```
Out[26]:
```
0.685400003194809
```

Applying the trick

Now we apply the reparametrization trick described in the paper, that is valid both for Conv and for FC layers, modifying weights and biases using the parameters in the BN layer so that the Conv/FC reparametrized absorbs the operations in BN layer:

In [27]:
```
def ReparamTrickBN(W, b, beta, gamma, mu, sigma2):
    W_rep = W * gamma / np.sqrt(sigma2)
    b_rep = (b - mu) * gamma / np.sqrt(sigma2) + beta
    return W_rep, b_rep
```

First we apply it to Conv1:

In [28]:
```
W_rep_c1, b_rep_c1 = ReparamTrickBN(
    W=allVariables['Model/W_c1:0'],
    b=allVariables['Model/b_c1:0'],
    beta=allVariables['BatchNorm/beta:0'],
    gamma=allVariables['BatchNorm/gamma:0'],
    mu=allVariables['BatchNorm/moving_mean:0'],
    sigma2=allVariables['BatchNorm/moving_variance:0'],
)
```

And then to FC1:

In [29]:
```
W_rep_fc1, b_rep_fc1 = ReparamTrickBN(
    W=allVariables['Model/W_fc1:0'],
    b=allVariables['Model/b_fc1:0'],
    beta=allVariables['BatchNorm_1/beta:0'],
    gamma=allVariables['BatchNorm_1/gamma:0'],
    mu=allVariables['BatchNorm_1/moving_mean:0'],
    sigma2=allVariables['BatchNorm_1/moving_variance:0'],
)
```

Since we now have the reparametrized Weights and Biases, we can finally load them into the CNN without BN:

In [31]:
```
NoBNcnn.sess.run(NoBNcnn.W_c1.assign(W_rep_c1))
NoBNcnn.sess.run(NoBNcnn.b_c1.assign(b_rep_c1))
NoBNcnn.sess.run(NoBNcnn.W_fc1.assign(W_rep_fc1))
NoBNcnn.sess.run(NoBNcnn.b_fc1.assign(b_rep_fc1))
NoBNcnn.sess.run(NoBNcnn.W_fc2.assign(allVariables['Model/W_fc2:0']))
NoBNcnn.sess.run(NoBNcnn.b_fc2.assign(allVariables['Model/b_fc2:0']))
```

Out[31]:
```
array([-0.0361975 ,  0.0180952 ,  0.08474635,  0.11768568,  0.18771696,
        0.05552447, -0.06432807,  0.1885533 ,  0.31808642,  0.06068037],
      dtype=float32)
```

At this point we save the model to be able to reproduce our results if desired.

In [32]:
```
NoBNcnn.save('Models/NoBN_tinyCNN')
```

```
INFO: TF Model saved in file: Models/NoBN_tinyCNN
```

Proving same accuracy

With our recently loaded reparametrized Weights and Biases in Conv1 and FC1, we evaluate test accuracy once again:

In [33]:
```
NoBNcnn.benchmark('TEST')
```

Out[33]:
```
0.9902000010013581
```

The final test accuracy for the CNN without BN is **99.02%**, which is the exact **same value** that the CNN with BN yielded (see section 4). We successfully modified the Weights and Biases of Conv and FC layers to absorb the subsequent BN layers.

This serves as proof of the reparametrization trick for Batch Normalization. QED

EXTRA: Performance gain

In [2]:
```
from CNNet import *
```

In [36]:
```
cnn = CNNet('tinyCNN', lr=0.01, loadInstance='Models/tinyCNN')
```

```
INFO:tensorflow:Restoring parameters from Models/tinyCNN
```

In [38]:
```
%timeit -r 30 cnn.predict(mnist.test.images[0:1000])
```

```
48.1 ms ± 77.5 µs per loop (mean ± std. dev. of 30 runs, 10 loops each)
```

In [33]:
```
NoBNcnn = CNNet('NoBN_tinyCNN', lr=0.01, flag_bNorm=False, loadInstance='Models/NoBN
_tinyCNN')
```

```
INFO:tensorflow:Restoring parameters from Models/NoBN_tinyCNN
```

In [35]:
```
%timeit -r 30 NoBNcnn.predict(mnist.test.images[0:1000])
```

```
38.2 ms ± 160 µs per loop (mean ± std. dev. of 30 runs, 10 loops each)
```

For this case, we have improved the performance from 48.1ms to 38.2ms by applying our trick, a **20% performance boost without loosing any accuracy**.