Optimization and Acceleration of Deep Learning on Various Hardware Platforms - Project Part3

Nitesh Bharadwaj Gundavarapu, A53317353

May 2020

1 Goal

We wish to compress CNNs in terms of memory and computational complexity. We use Tucker decomposition on convolutional layers to get a low-rank approximation and present the impact on accuracy.

2 Tucker Decomposition

Tucker decomposition is a higher order Singular Value Decomposition technique to factorize tensors. By choosing the principal tensors similar to PCA, the convolutional layer's weight matrix $W^{k\times k\times c\times f}$ is decomposed into a core matrix $k \times k \times R1 \times R2$, input matrix $c \times R1$ and output matrix $R2 \times f$.

$$\hat{W} = I \times C \times O$$

In Keras, we can implement I and O as 1×1 convolution layers and C as $k \times k$. Stride and padding from W are mapped to C while bias and activation are mapped to O.

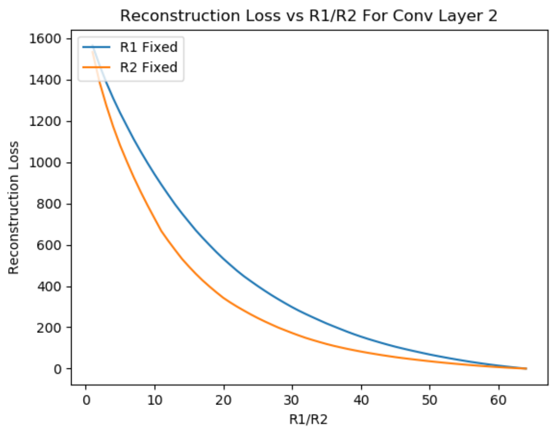
3 Complexity

Assuming stride of 1 and valid padding. The 1×1 convolutions don't change the input shape and only the # of channels change. So, we can simply compare the matrix sizes as the input shape scales the computations similarly for simple convolution and decomposed convolution.

Simple convolution $k\times k\times c\times f$ Decomposed convolution $c\times R1+k\times k\times R1\times R2+R2\times f$. So, if $R1$ and $R2$ are small, the amount of computations in second case can be very small. This is similar to taking a low rank PCA that reduces the amount of elements in a matrix and subsequent matrix multiplications are reduced in computational complexity.

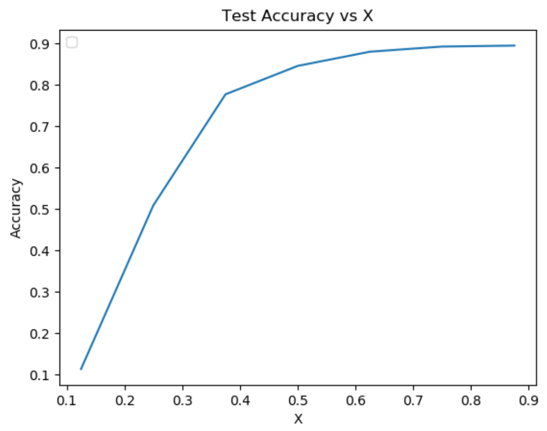
4 Empirical Evaluation

Tucker decomposition is performed on the convolutional layer's weight matrix W and using above idea of 11 convolutions a low rank approximation $\hat{W}$ is obtained. The reconstruction error is plotted in below figure upon varying the rank of approximation from $1 \rightarrow 64$. Results are presented below:



We can clearly see that reducing rank increases reconstruction error and vice-versa. But we can take advantage of the flat region of the curve from 30 to 60 without a blow up in reconstruction error. The trend is same for both dimensions.

In the next figure we analyze the change in accuracy by increasing the rank of approximation. This is performed on all layers of the network.



It is interesting to note a large flat area on the curve where we can reduce each the rank of each filter of each convolutional layer by a factor of 0.5 without a significant drop in performance. Note that reducing $R1$ by a half and $R2$ by a half, the computation of core reduces by 75%.

5 Appendix:

Below is the Python3 code:

```
import keras
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, Model
from keras.layers import Dense, Activation, Input, Lambda, Conv2D, MaxPooling2D, Flatten, Dr
from keras.layers.normalization import BatchNormalization
from tensorly.decomposition import partial_tucker
from keras import backend as K
from keras.engine.topology import Layer
import numpy as np

from keras.datasets import cifar10
from keras.utils import np_utils
from matplotlib import pyplot as plt
from utils import *
import tensorly as tl

(X_train, y_train), (X_test, y_test) = cifar10.load_data()

X_train=X_train.astype(np.float32)
X_test=X_test.astype(np.float32)
Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)
X_train /= 255
X_test /= 255
X_train=2*X_train-1
X_test=2*X_test-1


print('X_train shape:', X_train.shape)
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')

batch_size=100
lr=0.001
Training=True
Compressing=False
def get_model():
batch_norm_alpha=0.9
```

```
batch_norm_eps=1e-4

model=Sequential()

model.add(Conv2D(filters=64, kernel_size=3, strides=(1, 1), padding='valid',input_shape=[32,
model.add(Activation('relu'))
model.add(BatchNormalization(axis=-1, momentum=batch_norm_alpha, epsilon=batch_norm_eps))
model.add(Conv2D(filters=64, kernel_size=3, strides=(1, 1), padding='valid'))
model.add(Activation('relu'))
model.add(BatchNormalization(axis=-1, momentum=batch_norm_alpha, epsilon=batch_norm_eps))
model.add(MaxPooling2D(pool_size=(2, 2),strides=(2,2)))

model.add(Conv2D(filters=128, kernel_size=3, strides=(1, 1), padding='valid'))
model.add(Activation('relu'))
model.add(BatchNormalization(axis=-1, momentum=batch_norm_alpha, epsilon=batch_norm_eps))
model.add(Conv2D(filters=128, kernel_size=3, strides=(1, 1), padding='valid'))
model.add(Activation('relu'))
model.add(BatchNormalization(axis=-1, momentum=batch_norm_alpha, epsilon=batch_norm_eps))
model.add(MaxPooling2D(pool_size=(2, 2),strides=(2,2)))

model.add(Conv2D(filters=256, kernel_size=3, strides=(1, 1), padding='valid'))
model.add(Activation('relu'))
model.add(BatchNormalization(axis=-1, momentum=batch_norm_alpha, epsilon=batch_norm_eps))
model.add(Conv2D(filters=256, kernel_size=3, strides=(1, 1), padding='valid'))
model.add(Activation('relu'))
model.add(BatchNormalization(axis=-1, momentum=batch_norm_alpha, epsilon=batch_norm_eps))
#model.add(MaxPooling2D(pool_size=(2, 2),strides=(2,2)))

model.add(Flatten())

model.add(Dense(512))
model.add(Activation('relu'))
model.add(BatchNormalization(axis=-1, momentum=batch_norm_alpha, epsilon=batch_norm_eps))
model.add(Dense(512))
model.add(Activation('relu'))
model.add(BatchNormalization(axis=-1, momentum=batch_norm_alpha, epsilon=batch_norm_eps))
model.add(Dense(10))
model.add(Activation('softmax'))

return model

model=get_model()

weights_path='pretrained_cifar10.h5'
model.load_weights(weights_path)
opt = keras.optimizers.Adam(lr=0.001,decay=1e-6)
```

```
#complie the model with sparse categorical crossentropy loss function as you did in part 1
model.compile(loss=keras.losses.categorical_crossentropy,
  optimizer=opt,
  metrics=['accuracy'])
#make sure weights are loaded correctly by evaluating the model here and printing the outpu
# Step 1: Check accuracy
#evaluation =model.evaluate(X_test, Y_test) #TODO
#print('Loss Pretrained {} Acc Pretrained {}'.format(evaluation[0], evaluation[1])) #TODO


# Decomposition function Step 2
def decompose(tensor, r1, r2):
# tensor is KxKxcxf
# r1 in c dir, r2 in f direction
core, tucker_factors = partial_tucker(tensor, modes=[2,3], ranks=[r1,r2])
return  core, tucker_factors


def reconstruct(core, tucker_factors):
recon_w = core
recon_w = tl.tenalg.mode_dot(recon_w, tucker_factors[0], 2)
recon_w = tl.tenalg.mode_dot(recon_w, tucker_factors[1], 3)
return recon_w


def plot(results_map, xlabel, ylabel, title, filename):
legend = []
plt.figure()
for result_key in results_map:
result = results_map[result_key]
xvals = result['x']
yvals = result['y']
res_name = result['res_name']
plt.plot(xvals, yvals)
if res_name:
legend.append(res_name)
plt.title(title)
plt.ylabel(ylabel)
plt.xlabel(xlabel)
plt.legend(legend, loc='upper left')
plt.savefig(filename)


def reconstruction_loss(layer, r1, r2):
w = layer.get_weights()[0].copy()
core, tucker_factors = decompose(w, r1, r2)
```

```
recon_w = reconstruct(core, tucker_factors)
return np.linalg.norm(w-recon_w)**2

layer = model.layers[3]
r1_fix_res = [reconstruction_loss(model.layers[3],64,r) for r in range(1,65)]
r2_fix_res = [reconstruction_loss(model.layers[3],r,64) for r in range(1,65)]
res_map_r1r2 = {
'r1_fix': { 'x': range(1,65),
'y': r1_fix_res,
'res_name': 'R1 Fixed'},
'r2_fix': { 'x': range(1,65),
'y': r2_fix_res,
'res_name': 'R2 Fixed'}
}
plot(res_map_r1r2, 'R1/R2', 'Reconstruction Loss', 'Reconstruction Loss vs R1/R2 For Conv La


# Step3

def compress_network(model, x):
new_model = Sequential()
layers = [l for l in model.layers]
new_model.add(layers[0])
for i in range(1, len(layers)): # Skipping first conv
if isinstance(layers[i], keras.layers.Conv2D):
weights, biases = layers[i].get_weights()
f = weights.shape[3]
c = weights.shape[2]
R1 = int(x*c)
R2 = int(x*f)
stride = layers[i].strides
padding = layers[i].padding

core_w, tucker_factors = partial_tucker(weights, modes=[2, 3], ranks=[R1,R2])

I_w = tucker_factors[0]
I = Conv2D(filters=I_w.shape[1], kernel_size=1, strides=(1, 1), padding='valid', use_bias=Fa
new_model.add(I)
I_w = np.expand_dims(I_w,0)
I_w = np.expand_dims(I_w, 0)
I.set_weights([I_w])

core = Conv2D(filters=core_w.shape[-1], kernel_size=core_w.shape[0], strides=stride, padding
use_bias=False)
new_model.add(core)
core.set_weights([core_w])
```

```
O_w = tucker_factors[1]
O = Conv2D(filters=O_w.shape[0], kernel_size=1, strides=(1, 1), padding='valid', use_bias=Tr
O_w = np.transpose(O_w)
O_w = np.expand_dims(O_w,0)
O_w = np.expand_dims(O_w, 0)
new_model.add(O)
O.set_weights([O_w, biases])

else:
new_model.add(layers[i])

# pruned_model = Model(input=layers[0].input, output=x)
return new_model


def get_acc(model, x):
compressed = compress_network(model, x)
opt = keras.optimizers.Adam(lr=0.001, decay=1e-6)
compressed.compile(loss=keras.losses.categorical_crossentropy,
   optimizer=opt,
   metrics=['accuracy'])

evaluation = compressed.evaluate(X_test, Y_test)
return evaluation[1]


accs = [get_acc(model,x*1.0/8) for x in range(1,8)]
res = {
'acc_res':{
'x': [x*1.0/8 for x in range(1,8)],
'y': accs,
'res_name': ''
}
}
plot(res,'X','Accuracy', 'Test Accuracy vs X', 'test_acc.png')
```