Individual Final Report

Prepared by:

Ruonan Jia

Table of contents

Introduction

I was working on building model at tensorflow for original 100 classes dataset. For our dataset from kaggle, has the training data and test data, which test data is in HDF5 file. I split data as 0.8:0.2, training dataset and validation dataset. Check how well the model work and then save the model test it in the kaggle testing dataset.

Individual work

TensorFlow

I used original data from kaggle and resize it to 128*128, designed 4 layers. Test on 5 categories.

Model Summary
Tensorflow Framework
4 Convolution/MaxPooling layers & 2 Fully Connected Layer with dropout layer
Number of layers & size of kernels limited by input image size
Kernel size = 5 for first 2 convolution layers, 3 for the last 2 convolution layers, 2 for all pooling layer
Numbers of kernels: ranging from 64 to 128 for each layer
Batch size: 64
Epochs: 30
Cross Entropy Loss
Adam optimizer

Network / Model Design for pytorch

There is no generalized solution for designing a CNN. The design depends on the purpose of our project, like input characteristics, accuracy, training time, adaptation, computing resources ... We are focused on getting the highest accuracy. To begin, we approach the problem in an empirical way: initialize the network parameters with a lot of different numbers and adjust them to see what improves accuracy.

Things in model that affect our accuracy:

Number of Layers:

Number of layers	2 layers, kernel size 3	4 layers, kernel size 3	6 layers, kernel size 3
Accuracy	6%	~40%	60%

With a data size of [48000, 3, 128, 128] (48,000 images of 3 color channels of 128x128 pixels), more than 6 layers will give us the best accuracy. After the 6th Convolution/Max Pooling layer, we reach the final output size of 2x2, a significant reduction in size from the 128x128 input images. Achieving 60% accuracy on the test set is a big improvement, so we decided that 6 layers are necessary in our model.

Kernel Size: With a small kernel size, the kernel can capture the outline of the images better. Imagining we have a very big kernel size, when the kernel slide on the picture, we may miss some features. With an 128x128 input size, we decided to try kernel size 5x5 and 10x10 at the first four layers, 4x4 at 5th layer and 2x2 at 6th layer. The reason we use a much smaller kernel size at the last two layers is because size of the feature map is getting smaller by going through the MaxPooling layer (with a kernel size 2x2, stride=2, padding=1, the size of feature map after pooling is half of the input size from convolution layer). Kernel size 5 gets slightly higher accuracy than kernel size 10. We decided to use kernel size 10, which is able to see the pattern of feature map better than smaller kernel size.

The calculation of output size is a big part for designing a model by Pytorch.
We are using MaxPooling layer with a kernel size 2x2, stride=2, padding=1, which can give us the half of the input size from convolution layer. We want our output size from each layer is 64x64, 32x32…. To make this happen, we need to make the output size from convolution layer close to the input size from each layer. Then we decided to use padding=4 and stride =1, so we keep almost the same size as the input size.

See the example calculations as below:
Equations to calculate output size after convolution:

w = (w + 2*PAD−kernel size) / STRIDE +1
h = (h + 2*PAD−kernel size) / STRIDE +1

1st Convolution layer(kernel size=10, padding=4, stride=1)
1st MaxPolling layer(kernel size=2, padding=1, stride=2)

Conv1: w=(128+2*4-10)/1+1=127 h=(128+2*4-10)/1+1=127
Pool1: w=(127+1)/2=64 h=(127+1)/2=64

We use the same kernel size, padding size and stride size at the first 4 layer.
So the output size for each layer:
128 → 64 → 32 → 16 → 8
From the 5th layer, with an input size 8x8, we cannot use kernel size larger than 8. We picked kernel size as 4, at the same time, padding should not be bigger than kernel size.

5th Convolution layer (kernel size=4, padding=2, stride=1)
5th MaxPooling layer (kernel size=2, padding=1, stride=2)

Conv5: w=(8+2*2-4)/1+1=9 h=(8+2*2-4)/1+1=9

Pool5: w=(9+1)/2=5 h=(9+1)/2=5

6th Convolution layer(kernel size=2, padding=0, stride=1)
6th MaxPooling layer(kernel size=2, padding=0, stride=2)

Conv6: w=(5+2*0-2)/1+1=4 h=(5+2*0-2)/1+1=4
Pool6: w=(4+0)/2=2 h=(4+0)/2=2

Number of Kernels:
For a better result, we are more willing to capture enough main features from the original input. Theoretically, we need a big number (like 256) as our first layer number of kernels. However, in reality if we put too many inputs at the first layer, when we use the cloud, we get an error that the gpu will be out of memory; even though we tried to put the batch size smaller it will be worse and time cost is too high. The highest number we can put at first layer is 128. To solve this problem, we decided to start with a reasonable number of kernels at the first layer (like 64), then increase the number of kernels in the following 3 layers and start decreasing the number of kernels at the last 2 layers (so we won't have too many neurons, also during that time for each feature map a smaller number of kernels will be enough to capture the main feature). The number of kernels used for each layer is 64, 128, 256, 394, 256, 128. While the total number of feature maps created will be increasing, we also tried to have a decreasing list of number of kernel for each layer. The highest number we can start with is 128, so we tried 128, 128, 128, 128, 128, 64. The accuracy is 4% less than our model.

Dropout layer:
With the new dataset having around 4,000 images per class and 48,000 total images, it is difficult to have underfitting in neural network model. Overfitting happens when the model begins to track variance in the data, so we added dropout on the fully connected layer with 0.5 dropout ratio (drop half of the neurons) to prevent overfitting in our CNN model.

SoftMax function at last layer:
Our goal is to do images classification for 12 classes. For more than 2 classes classification, SoftMax function suits our model. The output of the model is a vector of length 12 producing values ranging from 0 to 1, representing the probability that the observation is of a certain class. The class with the highest such probability is the predicted output of the network.

Adam optimizer:
We used this training function because it showed better performance than SGD. SGD uses a static learning rate, while Adam is using dynamic learning rate. When the optimizer reaches the local minimum, the momentum will make it keeping going and find the global minimum. Adam will work better for a complicated neural network like CNN.

Epochs:
For the first trial, we pick an epoch number as 50, which cost us 5 hours to train the model. The training loss is very low from 20 epochs.

Model Summary
Pytorch Framework
6 Convolution/MaxPooling layers & 1 Fully Connected Layer with dropout layer
Number of layers & size of kernels limited by input image size
Kernel size = 10 for first 4 layers, 4 for 5th layer, 2 for last convolution layer
Numbers of kernels: ranging from 64 to 384 for each layer
Batch size: 40
Epochs: 25
Cross Entropy Loss
Adam optimizer

Python Script Draw CNN Diagram

For understanding the CNN better, I decided to draw a CNN Diagram either windows visio or by hand to show the input size, output size, and the CNN part

The portion I did on the project in detail

Tensorflow script cnn2.py

```
# -*- coding: utf-8 -*-
# delete by hand the wrong pics
from skimage import io, transform
import glob
import os
import tensorflow as tf
import numpy as np
import time

path = '/home/ubuntu/Deep-Learning/cnn/data/'

#resize=100*100
w=128
h=128
c=3


#read images
def read_img(path):
    cate=[path+x for x in os.listdir(path) if os.path.isdir(path+x)]
    imgs=[]
    labels=[]
    for idx,folder in enumerate(cate):
        for im in glob.glob(folder+'/*.jpg'):
            # print('reading the images:%s'%(im))
            img=io.imread(im)
            img=transform.resize(img,(w,h))
            imgs.append(img)
            labels.append(idx)
    return np.asarray(imgs,np.float32),np.asarray(labels,np.int32)
data,label=read_img(path)
```

```
# shuffle
num_example=data.shape[0]
arr=np.arange(num_example)
np.random.shuffle(arr)
data=data[arr]
label=label[arr]


# split training set and test set
ratio=0.8
s=np.int(num_example*ratio)
x_train=data[:s]
y_train=label[:s]
x_val=data[s:]
y_val=label[s:]

# -----------------bengin of network----------------------
# 占位符placeholder shape
x=tf.placeholder(tf.float32,shape=[None,w,h,c],name='x')
y_=tf.placeholder(tf.int32,shape=[None,],name='y_')

# first layer (128——>64)
conv1=tf.layers.conv2d(
      inputs=x,
      filters=32,
      kerne_size=[5, 5],
      padding="same",
      activation=tf.nn.relu,
      kernel_initializer=tf.truncated_normal_initializer(stddev=0.01))
pool1=tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)

# second layer(64->32)
conv2=tf.layers.conv2d(
      inputs=pool1,
      filters=64,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu,
      kernel_initializer=tf.truncated_normal_initializer(stddev=0.01))
pool2=tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)

# third layer(32->16)
conv3=tf.layers.conv2d(
      inputs=pool2,
      filters=128,
      kernel_size=[3, 3],
      padding="same",
      activation=tf.nn.relu,
      kernel_initializer=tf.truncated_normal_initializer(stddev=0.01))
pool3=tf.layers.max_pooling2d(inputs=conv3, pool_size=[2, 2], strides=2)

# forth layer(16->8)
conv4=tf.layers.conv2d(
      inputs=pool3,
      filters=128,
      kernel_size=[3, 3],
      padding="same",
      activation=tf.nn.relu,
      kernel_initializer=tf.truncated_normal_initializer(stddev=0.01))
pool4=tf.layers.max_pooling2d(inputs=conv4, pool_size=[2, 2], strides=2)
```

```
conv5=tf.layers.conv2d(
      inputs=pool4,
      filters=128,
      kernel_size=[2, 2],
      padding="same",
      activation=tf.nn.relu,
      kernel_initializer=tf.truncated_normal_initializer(stddev=0.01))
pool5=tf.layers.max_pooling2d(inputs=conv5, pool_size=[2, 2], strides=2)


re1 = tf.reshape(pool5, [-1, 8 * 8 * 128])
# explaining for presentation

# last inner layer
dense1 = tf.layers.dense(inputs=re1,
                      units =1024,  # feature maps size*feature maps number 128*
                      activation= tf.nn.relu,
                      kernel_initializer
=tf.truncated_normal_initializer(stddev=0.01),
                      kernel_regularizer =tf.contrib.layers.l2_regularizer(0.003))
dense2= tf.layers.dense(inputs=dense1,
                      units=512,
                      activation=tf.nn.relu,
                      kernel_initializer=tf.truncated_normal_initializer(stddev=0.01),
                      kernel_regularizer=tf.contrib.layers.l2_regularizer(0.003))
logits= tf.layers.dense(inputs=dense2,
                        units=5,
                        activation=None,

kernel_initializer=tf.truncated_normal_initializer(stddev=0.01),
                        kernel_regularizer=tf.contrib.layers.l2_regularizer(0.003))
# ---------------------------end of network---------------------------

loss=tf.losses.sparse_softmax_cross_entropy(labels=y_,logits=logits)
train_op=tf.train.AdamOptimizer(learning_rate=0.001).minimize(loss)
correct_prediction = tf.equal(tf.cast(tf.argmax(logits,1),tf.int32), y_)
acc= tf.reduce_mean(tf.cast(correct_prediction, tf.float32))


# build minibatch dataset
def minibatches(inputs=None, targets=None, batch_size=None, shuffle=False):
    assert len(inputs) == len(targets)
    if shuffle:
        indices = np.arange(len(inputs))
        np.random.shuffle(indices)
    for start_idx in range(0, len(inputs) - batch_size + 1, batch_size):
        if shuffle:
            excerpt = indices[start_idx:start_idx + batch_size]
        else:
            excerpt = slice(start_idx, start_idx + batch_size)
        yield inputs[excerpt], targets[excerpt]


# train and test

n_epoch=30
batch_size=64
sess=tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
for epoch in range(n_epoch):
```

```
    start_time = time.time()

    #training
    train_loss, train_acc, n_batch = 0, 0, 0
    for x_train_a, y_train_a in minibatches(x_train, y_train, batch_size,
shuffle=True):
        _,err,ac=sess.run([train_op,loss,acc], feed_dict={x: x_train_a, y_:
y_train_a})
        train_loss += err; train_acc += ac; n_batch += 1
    print("   train loss: %f" % (train_loss/ n_batch))
    print("   train acc: %f" % (train_acc/ n_batch))

    #validation
    val_loss, val_acc, n_batch = 0, 0, 0
    for x_val_a, y_val_a in minibatches(x_val, y_val, batch_size, shuffle=False):
        err, ac = sess.run([loss,acc], feed_dict={x: x_val_a, y_: y_val_a})
        val_loss += err; val_acc += ac; n_batch += 1
    print("   validation loss: %f" % (val_loss/ n_batch))
    print("   validation acc: %f" % (val_acc/ n_batch))

sess.close()
```

DRAW.py

```
import os
import numpy as np
import matplotlib.pyplot as plt
plt.rcdefaults()
from matplotlib.lines import Line2D
from matplotlib.patches import Rectangle
from matplotlib.collections import PatchCollection


NumConvMax = 8
NumFcMax = 20
White = 1.
Light = 0.7
Medium = 0.5
Dark = 0.3
Black = 0.


def add_layer(patches, colors, size=24, num=13,
              top_left=[0, 0],
              loc_diff=[3, -3],
              ):
    # add a rectangle
    top_left = np.array(top_left)
    loc_diff = np.array(loc_diff)
    loc_start = top_left - np.array([0, size])
    for ind in range(num):
        patches.append(Rectangle(loc_start + ind * loc_diff, size, size))
        if ind % 2:
            colors.append(Medium)
        else:
            colors.append(Light)


def add_mapping(patches, colors, start_ratio, patch_size, ind_bgn,
                top_left_list, loc_diff_list, num_show_list, size_list):

    start_loc = top_left_list[ind_bgn] \
```

```
        + (num_show_list[ind_bgn] - 1) * np.array(loc_diff_list[ind_bgn]) \
        + np.array([start_ratio[0] * size_list[ind_bgn],
                    -start_ratio[1] * size_list[ind_bgn]])

    end_loc = top_left_list[ind_bgn + 1] \
        + (num_show_list[ind_bgn + 1] - 1) \
        * np.array(loc_diff_list[ind_bgn + 1]) \
        + np.array([(start_ratio[0] + .5 * patch_size / size_list[ind_bgn]) *
                    size_list[ind_bgn + 1],
                    -(start_ratio[1] - .5 * patch_size / size_list[ind_bgn]) *
                    size_list[ind_bgn + 1]])

    patches.append(Rectangle(start_loc, patch_size, patch_size))
    colors.append(Dark)
    patches.append(Line2D([start_loc[0], end_loc[0]],
                          [start_loc[1], end_loc[1]]))
    colors.append(Black)
    patches.append(Line2D([start_loc[0] + patch_size, end_loc[0]],
                          [start_loc[1], end_loc[1]]))
    colors.append(Black)
    patches.append(Line2D([start_loc[0], end_loc[0]],
                          [start_loc[1] + patch_size, end_loc[1]]))
    colors.append(Black)
    patches.append(Line2D([start_loc[0] + patch_size, end_loc[0]],
                          [start_loc[1] + patch_size, end_loc[1]]))
    colors.append(Black)


def label(xy, text, xy_off=[0, 4]):
    plt.text(xy[0] + xy_off[0], xy[1] + xy_off[1], text,
             family='sans-serif', size=8)


if __name__ == '__main__':

    fc_unit_size = 2
    layer_width = 100

    patches = []
    colors = []

    fig, ax = plt.subplots()


    ############################
    # conv layers
    size_list = [128, 127, 64, 63, 32, 31, 16, 15, 8, 9, 5, 4, 2]
    num_list = [3, 64, 64, 128, 128, 256, 256, 384, 384, 256, 256, 128, 128]
    x_diff_list = [0, layer_width, layer_width, layer_width, layer_width, layer_width,
layer_width, layer_width, layer_width, layer_width, layer_width, layer_width,
layer_width]
    text_list = ['Inputs'] + ['Feature\nmaps'] * (len(size_list) - 1)
    loc_diff_list = [[3, -3]] * len(size_list)

    num_show_list = list(map(min, num_list, [NumConvMax] * len(num_list)))
    top_left_list = np.c_[np.cumsum(x_diff_list), np.zeros(len(x_diff_list))]

    for ind in range(len(size_list)):
        add_layer(patches, colors, size=size_list[ind],
                  num=num_show_list[ind],
                  top_left=top_left_list[ind], loc_diff=loc_diff_list[ind])
```

```
        label(top_left_list[ind], text_list[ind] + '\n{}@{}x{}'.format(
            num_list[ind], size_list[ind], size_list[ind]))


    ############################
    # in between layers
    start_ratio_list = [[0.4, 0.5], [0.4, 0.8], [0.4, 0.5], [0.4, 0.8], [0.4, 0.5],
[0.4, 0.8], [0.4, 0.5], [0.4, 0.8], [0.4, 0.5], [0.4, 0.8], [0.4, 0.5], [0.4, 0.8]]
    patch_size_list = [10, 2, 10, 2, 10, 2, 10, 2, 4, 2, 2, 2]
    ind_bgn_list = range(len(patch_size_list))
    text_list = ['Convolution', 'Max-pooling', 'Convolution', 'Max-pooling',
'Convolution', 'Max-pooling', 'Convolution', 'Max-pooling','Convolution', 'Max-
pooling', 'Convolution', 'Max-pooling']

    for ind in range(len(patch_size_list)):
        add_mapping(patches, colors, start_ratio_list[ind],
                    patch_size_list[ind], ind,
                    top_left_list, loc_diff_list, num_show_list, size_list)
        label(top_left_list[ind], text_list[ind] + '\n{}x{} kernel'.format(
            patch_size_list[ind], patch_size_list[ind]), xy_off=[26, -65])


    ############################
    # fully connected layers
    size_list = [fc_unit_size, fc_unit_size]
    num_list = [512, 12]
    num_show_list = list(map(min, num_list, [NumFcMax] * len(num_list)))
    x_diff_list = [sum(x_diff_list) + layer_width, layer_width, layer_width]
    top_left_list = np.c_[np.cumsum(x_diff_list), np.zeros(len(x_diff_list))]
    loc_diff_list = [[fc_unit_size, -fc_unit_size]] * len(top_left_list)
    text_list = ['Hidden\nunits'] * (len(size_list) - 1) + ['Outputs']

    for ind in range(len(size_list)):
        add_layer(patches, colors, size=size_list[ind], num=num_show_list[ind],
                  top_left=top_left_list[ind], loc_diff=loc_diff_list[ind])
        label(top_left_list[ind], text_list[ind] + '\n{}'.format(
            num_list[ind]))

    text_list = ['Flatten\n', 'Fully\nconnected', 'Fully\nconnected']

    for ind in range(len(size_list)):
        label(top_left_list[ind], text_list[ind], xy_off=[-10, -65])

    ############################
    colors += [0, 1]
    collection = PatchCollection(patches, cmap=plt.cm.gray)
    collection.set_array(np.array(colors))
    ax.add_collection(collection)
    plt.tight_layout()
    plt.axis('equal')
    plt.axis('off')
    plt.show()
    fig.set_size_inches(8, 2.5)

    fig_dir = './'
    fig_ext = '.png'
    fig.savefig(os.path.join(fig_dir, 'convnet_fig' + fig_ext),
                bbox_inches='tight', pad_inches=0)
```

Result

With 5 classes, 4 layers I got 63% accuracy; with 25 classes, 4 layers I got 36% accuracy.

```
validation loss: 1.626814
validation acc: 0.618750
train loss: 0.182965
train acc: 0.939516
validation loss: 1.973666
validation acc: 0.608333
train loss: 0.185969
train acc: 0.936240
validation loss: 1.811569
validation acc: 0.634375
train loss: 0.117678
train acc: 0.962198
validation loss: 2.233531
```

Summary and Conclusion

For classified 100 classes, 1000 images per class definitely not going to work. The input size also could be bigger like 500x500.
Images Pre-Processing will be very important: for food image, the color, outline and texture will be important. To maintain the same shape of the food, I should padding before resize keep the shape same proportion. And also do random cropping and flipping can help to generate significantly more data. Also the cost of time and memory of GPU will be another issue.

Tools: SIFT (Scale-Invariant Feature Transform) / Open CV
Approach: Outline of the food Color Descriptor; Extract effective information (take out the plate or table in the image)

CNN parameters: Kernel size (smaller like 5 or 3); Number of Kernels(from high to low); Batch Normalization

Post-Processing:

work on the lowest accuracy category (relabel or go)
More than 1 class in one picture

The percentage of the code I found from the internet

cnn2.py
(151-138)/(151+20)=7.6%

DRAW.py
(153-43)/153=71%

Reference

Python Script for illustrating CNN
https://github.com/yu4u/convnet-drawer/blob/master/convnet_drawer.py