# homework7

April 2, 2019

## 1 Homework 7 - Berkeley STAT 157

**Your name: XX, SID YY, teammates A,B,C** (Please add your name, SID and teammates to ease Ryan and Rachel to grade.)

**Please submit your homework through gradescope**

Handout 4/2/2019, due 4/9/2019 by 4pm.

This homework deals with fine-tuning for computer vision. In this task, we attempt to identify 120 different breeds of dogs. The data set used in this competition is actually a subset of the ImageNet data set. Different from the images in the CIFAR-10 data set used in the previous homework, the images in the ImageNet data set are higher and wider and their dimensions are inconsistent. Again, you need to use GPU.

The dataset is available at Kaggle. The rule is similar to homework 6:

- work as a team
- submit your results into Kaggle
- take a screen shot of your best score and insert it below
- the top 3 teams/individuals will be awarded with 500 dollar AWS credits

First, import the packages or modules required for the competition.

```
In [1]: import collections
        import d2l
        import math
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import model_zoo, nn
        from mxnet.gluon import data as gdata, loss as gloss, utils as gutils
        import os
        import shutil
        import time
        import zipfile
```

### 1.1 Obtain and Organize the Data Sets

The competition data is divided into a training set and testing set. The training set contains 10,222 images and the testing set contains 10,357 images. The images in both sets are in JPEG format. These images contain three RGB channels (color) and they have different heights and widths. There are 120 breeds of dogs in the training set, including Labradors, Poodles, Dachshunds, Samoyeds, Huskies, Chihuahuas, and Yorkshire Terriers.

### 1.1.1 Download the Data Set

After logging in to Kaggle, we can click on the "Data" tab on the dog breed identification competition webpage shown in Figure 9.17 and download the training data set "train.zip", the testing data set "test.zip", and the training data set labels "label.csv.zip". After downloading the files, place them in the three paths below:

- kaggle_dog/train.zip
- kaggle_dog/test.zip
- kaggle_dog/labels.csv.zip

To make it easier to get started, we provide a small-scale sample of the data set mentioned above, "train_valid_test_tiny.zip". If you are going to use the full data set for the Kaggle competition, you will also need to change the `demo` variable below to `False`.

```
In [2]: # If you use the full data set downloaded for the Kaggle competition,
        # change the variable below to False.
        demo = True
        data_dir = './kaggle_dog'
        if demo:
            if not os.path.exists(data_dir):
                os.mkdir(data_dir)
            gutils.download('https://github.com/d2l-ai/d2l-en/raw/master/data/kaggle_dog/train_
                            data_dir)
            zipfiles = ['train_valid_test_tiny.zip']
        else:
            zipfiles = ['train.zip', 'test.zip', 'labels.csv.zip']
        for f in zipfiles:
            with zipfile.ZipFile(data_dir + '/' + f, 'r') as z:
                z.extractall(data_dir)
```

### 1.1.2 Organize the Data Set

Next, we define the `reorg_train_valid` function to segment the validation set from the original Kaggle competition training set. The parameter `valid_ratio` in this function is the ratio of the number of examples of each dog breed in the validation set to the number of examples of the breed with the least examples (66) in the original training set. After organizing the data, images of the same breed will be placed in the same folder so that we can read them later.

```
In [3]: def reorg_train_valid(data_dir, train_dir, input_dir, valid_ratio, idx_label):
            # The number of examples of the least represented breed in the training set.
            min_n_train_per_label = (
                collections.Counter(idx_label.values()).most_common()[:-2:-1][0][1])
            # The number of examples of each breed in the validation set.
            n_valid_per_label = math.floor(min_n_train_per_label * valid_ratio)
            label_count = {}
            for train_file in os.listdir(os.path.join(data_dir, train_dir)):
                idx = train_file.split('.')[0]
                label = idx_label[idx]
```

```
                d2l.mkdir_if_not_exist([data_dir, input_dir, 'train_valid', label])
                shutil.copy(os.path.join(data_dir, train_dir, train_file),
                            os.path.join(data_dir, input_dir, 'train_valid', label))
                if label not in label_count or label_count[label] < n_valid_per_label:
                    d2l.mkdir_if_not_exist([data_dir, input_dir, 'valid', label])
                    shutil.copy(os.path.join(data_dir, train_dir, train_file),
                                os.path.join(data_dir, input_dir, 'valid', label))
                    label_count[label] = label_count.get(label, 0) + 1
                else:
                    d2l.mkdir_if_not_exist([data_dir, input_dir, 'train', label])
                    shutil.copy(os.path.join(data_dir, train_dir, train_file),
                                os.path.join(data_dir, input_dir, 'train', label))
```

The `reorg_dog_data` function below is used to read the training data labels, segment the validation set, and organize the training set.

```
In [4]: def reorg_dog_data(data_dir, label_file, train_dir, test_dir, input_dir,
                           valid_ratio):
            # Read the training data labels.
            with open(os.path.join(data_dir, label_file), 'r') as f:
                # Skip the file header line (column name).
                lines = f.readlines()[1:]
                tokens = [l.rstrip().split(',') for l in lines]
                idx_label = dict(((idx, label) for idx, label in tokens))
            reorg_train_valid(data_dir, train_dir, input_dir, valid_ratio, idx_label)
            # Organize the training set.
            d2l.mkdir_if_not_exist([data_dir, input_dir, 'test', 'unknown'])
            for test_file in os.listdir(os.path.join(data_dir, test_dir)):
                shutil.copy(os.path.join(data_dir, test_dir, test_file),
                            os.path.join(data_dir, input_dir, 'test', 'unknown'))
```

Because we are using a small data set, we set the batch size to 1. During actual training and testing, we would use the entire Kaggle Competition data set and call the `reorg_dog_data` function to organize the data set. Likewise, we would need to set the `batch_size` to a larger integer, such as 128.

```
In [5]: if demo:
            # Note: Here, we use a small data set and the batch size should be set
            # smaller. When using the complete data set for the Kaggle competition,
            # we can set the batch size to a larger integer.
            input_dir, batch_size = 'train_valid_test_tiny', 1
        else:
            label_file, train_dir, test_dir = 'labels.csv', 'train', 'test'
            input_dir, batch_size, valid_ratio = 'train_valid_test', 128, 0.1
            reorg_dog_data(data_dir, label_file, train_dir, test_dir, input_dir,
                           valid_ratio)
```

## 1.2 Image Augmentation

The size of the images in this section are larger than the images in the previous section. Here are some more image augmentation operations that might be useful.

In [6]:
```
transform_train = gdata.vision.transforms.Compose([
    # Randomly crop the image to obtain an image with an area of 0.08 to 1
    # of the original area and height to width ratio between 3/4 and 4/3.
    # Then, scale the image to create a new image with a height and width
    # of 224 pixels each.
    gdata.vision.transforms.RandomResizedCrop(224, scale=(0.08, 1.0),
                                              ratio=(3.0/4.0, 4.0/3.0)),
    gdata.vision.transforms.RandomFlipLeftRight(),
    # Randomly change the brightness, contrast, and saturation.
    gdata.vision.transforms.RandomColorJitter(brightness=0.4, contrast=0.4,
                                              saturation=0.4),
    # Add random noise.
    gdata.vision.transforms.RandomLighting(0.1),
    gdata.vision.transforms.ToTensor(),
    # Standardize each channel of the image.
    gdata.vision.transforms.Normalize([0.485, 0.456, 0.406],
                                      [0.229, 0.224, 0.225])])
```

During testing, we only use definite image preprocessing operations.

In [7]:
```
transform_test = gdata.vision.transforms.Compose([
    gdata.vision.transforms.Resize(256),
    # Crop a square of 224 by 224 from the center of the image.
    gdata.vision.transforms.CenterCrop(224),
    gdata.vision.transforms.ToTensor(),
    gdata.vision.transforms.Normalize([0.485, 0.456, 0.406],
                                      [0.229, 0.224, 0.225])])
```

## 1.3 Read the Data Set

As in the previous section, we can create an `ImageFolderDataset` instance to read the data set containing the original image files.

In [8]:
```
train_ds = gdata.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'train'), flag=1)
valid_ds = gdata.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'valid'), flag=1)
train_valid_ds = gdata.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'train_valid'), flag=1)
test_ds = gdata.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'test'), flag=1)
```

Here, we create a `DataLoader` instance, just like in the previous section.

```
In [9]: train_iter = gdata.DataLoader(train_ds.transform_first(transform_train),
                                      batch_size, shuffle=True, last_batch='keep')
        valid_iter = gdata.DataLoader(valid_ds.transform_first(transform_test),
                                      batch_size, shuffle=True, last_batch='keep')
        train_valid_iter = gdata.DataLoader(train_valid_ds.transform_first(
            transform_train), batch_size, shuffle=True, last_batch='keep')
        test_iter = gdata.DataLoader(test_ds.transform_first(transform_test),
                                     batch_size, shuffle=False, last_batch='keep')
```

### 1.4 Define the Model

The data set for this competition is a subset of the ImageNet data set. Therefore, we can use the approach discussed in the "Fine Tuning" section to select a model pre-trained on the entire ImageNet data set and use it to extract image features to be input in the custom small-scale output network. Gluon provides a wide range of pre-trained models. Here, we will use the pre-trained ResNet-34 model. Because the competition data set is a subset of the pre-training data set, we simply reuse the input of the pre-trained model's output layer, i.e. the extracted features. Then, we can replace the original output layer with a small custom output network that can be trained, such as two fully connected layers in a series. Different from the experiment in the "Fine Tuning" section, here, we do not retrain the pre-trained model used for feature extraction. This reduces the training time and the memory required to store model parameter gradients.

You must note that, during image augmentation, we use the mean values and standard deviations of the three RGB channels for the entire ImageNet data set for normalization. This is consistent with the normalization of the pre-trained model.

```
In [10]: def get_net(ctx):
             finetune_net = model_zoo.vision.resnet34_v2(pretrained=True)
             # Define a new output network.
             finetune_net.output_new = nn.HybridSequential(prefix='')
             finetune_net.output_new.add(nn.Dense(256, activation='relu'))
             # There are 120 output categories.
             finetune_net.output_new.add(nn.Dense(120))
             # Initialize the output network.
             finetune_net.output_new.initialize(init.Xavier(), ctx=ctx)
             # Distribute the model parameters to the CPUs or GPUs used for computation.
             finetune_net.collect_params().reset_ctx(ctx)
             return finetune_net
```

When calculating the loss, we first use the member variable `features` to obtain the input of the pre-trained model's output layer, i.e. the extracted feature. Then, we use this feature as the input for our small custom output network and compute the output.

```
In [11]: loss = gloss.SoftmaxCrossEntropyLoss()

         def evaluate_loss(data_iter, net, ctx):
             l_sum, n = 0.0, 0
             for X, y in data_iter:
                 y = y.as_in_context(ctx)
                 output_features = net.features(X.as_in_context(ctx))
```

```
            outputs = net.output_new(output_features)
            l_sum += loss(outputs, y).sum().asscalar()
            n += y.size
        return l_sum / n
```

### 1.5 Define the Training Functions

We will select the model and tune hyper-parameters according to the model's performance on the validation set. The model training function `train` only trains the small custom output network.

```
In [12]: def train(net, train_iter, valid_iter, num_epochs, lr, wd, ctx, lr_period,
               lr_decay):
             # Only train the small custom output network.
             trainer = gluon.Trainer(net.output_new.collect_params(), 'sgd',
                                     {'learning_rate': lr, 'momentum': 0.9, 'wd': wd})
             for epoch in range(num_epochs):
                 train_l_sum, n, start = 0.0, 0, time.time()
                 if epoch > 0 and epoch % lr_period == 0:
                     trainer.set_learning_rate(trainer.learning_rate * lr_decay)
                 for X, y in train_iter:
                     y = y.as_in_context(ctx)
                     output_features = net.features(X.as_in_context(ctx))
                     with autograd.record():
                         outputs = net.output_new(output_features)
                         l = loss(outputs, y).sum()
                     l.backward()
                     trainer.step(batch_size)
                     train_l_sum += l.asscalar()
                     n += y.size
                 time_s = "time %.2f sec" % (time.time() - start)
                 if valid_iter is not None:
                     valid_loss = evaluate_loss(valid_iter, net, ctx)
                     epoch_s = ("epoch %d, train loss %f, valid loss %f, "
                                % (epoch + 1, train_l_sum / n, valid_loss))
                 else:
                     epoch_s = ("epoch %d, train loss %f, "
                                % (epoch + 1, train_l_sum / n))
                 print(epoch_s + time_s + ', lr ' + str(trainer.learning_rate))
```

### 1.6 Train and Validate the Model

Now, we can train and validate the model. The following hyper-parameters can be tuned. For example, we can increase the number of epochs. Because `lr_period` and `lr_decay` are set to 10 and 0.1 respectively, the learning rate of the optimization algorithm will be multiplied by 0.1 after every 10 epochs.

```
In [13]: ctx, num_epochs, lr, wd = d2l.try_gpu(), 1, 0.01, 1e-4
         lr_period, lr_decay, net = 10, 0.1, get_net(ctx)
```

```
net.hybridize()
train(net, train_iter, valid_iter, num_epochs, lr, wd, ctx, lr_period,
      lr_decay)
```

```
Downloading /Users/muli/.mxnet/models/resnet34_v2-9d6b80bb.zip from https://apache-mxnet.s3-ac
epoch 1, train loss 5.320887, valid loss 4.771201, time 8.65 sec, lr 0.01
```

## 1.7 Classify the Testing Set and Submit Results on Kaggle

After obtaining a satisfactory model design and hyper-parameters, we use all training data sets (including validation sets) to retrain the model and then classify the testing set. Note that predictions are made by the output network we just trained.

```
In [14]: net = get_net(ctx)
         net.hybridize()
         train(net, train_valid_iter, None, num_epochs, lr, wd, ctx, lr_period,
               lr_decay)

         preds = []
         for data, label in test_iter:
             output_features = net.features(data.as_in_context(ctx))
             output = nd.softmax(net.output_new(output_features))
             preds.extend(output.asnumpy())
         ids = sorted(os.listdir(os.path.join(data_dir, input_dir, 'test/unknown')))
         with open('submission.csv', 'w') as f:
             f.write('id,' + ','.join(train_valid_ds.synsets) + '\n')
             for i, output in zip(ids, preds):
                 f.write(i.split('.')[0] + ',' + ','.join(
                     [str(num) for num in output]) + '\n')
```

```
epoch 1, train loss 5.090321, time 18.22 sec, lr 0.01
```

After executing the above code, we will generate a “submission.csv” file. The format of this file is consistent with the Kaggle competition requirements.

## 1.8 Hints to Improve Your Results

- You should download the whole data set from Kaggle and switch to `demo=False`.
- Try to increase the `batch_size` (batch size) and `num_epochs` (number of epochs).
- Try a deeper pre-trained model, you may find models from gluoncv.