Coursework2: Convolutional Neural Networks

instructions

Please submit a version of this notebook containing your answers **together with your trained model** on CATe as CW2.zip. Write your answers in the cells below each question.

A PDF version of this notebook is also provided in case the figures do not render correctly.

The deadline for submission is 19:00, Thu 14th February, 2019

Setting up working environment

For this coursework you will need to train a large network, therefore we recommend you work with Google Colaboratory, which provides free GPU time. You will need a Google account to do so.

Please log in to your account and go to the following page: https://colab.research.google.com (https://colab.research.google.com). Then upload this notebook.

For GPU support, go to "Edit" -> "Notebook Settings", and select "Hardware accelerator" as "GPU".

You will need to install pytorch by running the following cell:

In [25]:
```
!pip install torch torchvision
```

```
Requirement already satisfied: torch in /anaconda3/envs/Math/lib/p
ython3.7/site-packages (1.0.0)
Requirement already satisfied: torchvision in /anaconda3/envs/Math
/lib/python3.7/site-packages (0.2.1)
Requirement already satisfied: numpy in /anaconda3/envs/Math/lib/p
ython3.7/site-packages (from torchvision) (1.15.4)
Requirement already satisfied: pillow>=4.1.1 in /anaconda3/envs/Ma
th/lib/python3.7/site-packages (from torchvision) (5.4.1)
Requirement already satisfied: six in /anaconda3/envs/Math/lib/pyt
hon3.7/site-packages (from torchvision) (1.12.0)
You are using pip version 19.0.1, however version 19.0.2 is availa
ble.
You should consider upgrading via the 'pip install --upgrade pip'
command.
```

Introduction

For this coursework you will implement one of the most commonly used model for image recognition tasks, the Residual Network. The architecture is introduced in 2015 by Kaiming He, et al. in the paper "Deep residual learning for image recognition" (https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.p

In a residual network, each block contains some convolutional layers, plus "skip" connections, which allow the activations to by pass a layer, and then be summed up with the activations of the skipped layer. The image below illustrates a building block in residual networks.

resnet-block

Depending on the number of building blocks, resnets can have different architectures, for example ResNet-50, ResNet-101 and etc. Here you are required to build ResNet-18 to perform classification on the CIFAR-10 dataset, therefore your network will have the following architecture:

resnet

Part 1 (40 points)

In this part, you will use basic pytorch operations to define the 2D convolution and max pooling operation.

YOUR TASK

- implement the forward pass for Conv2D and MaxPool2D
- You can only fill in the parts which are specified as "YOUR CODE HERE"
- You are **NOT** allowed to use the torch.nn module and the conv2d/maxpooling functions in torch.nn.functional

In [26]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

In [27]:
```
class Conv2D(nn.Module):

    def __init__(self, inchannel, outchannel, kernel_size, stride,
padding, bias = True):

        super(Conv2D, self).__init__()

        self.inchannel = inchannel
        self.outchannel = outchannel
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding

        self.weights = nn.Parameter(torch.Tensor(outchannel, inchan
nel,
                                                 kernel_size, kerne
l_size))
        self.weights.data.normal_(-0.1, 0.1)

        if bias:
            self.bias = nn.Parameter(torch.Tensor(outchannel, ))
            self.bias.data.normal_(-0.1, 0.1)
        else:
            self.bias = None


    def forward(self, x):

        ###########################################################
###
        #                        YOUR CODE HERE
#
        ###########################################################
###
        #input x[batch,channel,x,x]

        batch=x.shape[0]

        size=x.shape[2]
        x_output=int(((size+2*self.padding-self.kernel_size)/self.s
tride)+1)

        output=torch.tensor(()).new_zeros((batch,self.outchannel,x_
output,x_output),device='cuda')

        x_padding=torch.tensor(()).new_zeros((batch,self.inchannel,
size+2*self.padding,size+2*self.padding),device='cuda')
```

```
        x_unfold=x_padding.unfold(2,self.kernel_size,self.stride).u
nfold(3,self.kernel_size,self.stride)

        x_unfold=x_unfold.reshape(batch,x_output,x_output,self.inch
annel*self.kernel_size*self.kernel_size)

        weights=self.weights.reshape(self.outchannel,self.inchannel
*self.kernel_size*self.kernel_size).transpose(0,1)


        output=x_unfold@weights+self.bias if self.bias != None else
x_unfold@weights


        ###########################################################
###
        #                       END OF YOUR CODE
#
        ###########################################################
###


        return output
```

In [28]:
```
class MaxPool2D(nn.Module):

    def __init__(self, pooling_size):
        # assume pooling_size = kernel_size = stride

        super(MaxPool2D, self).__init__()

        self.pooling_size = pooling_size


    def forward(self, x):


        ############################################################
###
        #                          YOUR CODE HERE
#
        ############################################################
###
        #input x[batch,channel,x,x]
        batch=x.shape[0]
        x=x.shape[2]
        channel=x.shape[1]
        size=int((x -self.pooling_size)/self.pooling_size+1)
        output=torch.tensor(()).new_zeros(batch,channel,size,size)
        x_unfold=x.unfold(2, self.pooling_size, self.pooling_size).
unfold(3,self.pooling_size,self.pooling_size)
        x_unfold=x_unfold.reshape(batch,channel,x_unfold.shape[2],x
_unfold.shape[3],x_unfold.shape[4]*x_unfold.shape[5])

        output = torch.max(x_unfold,dim=4)[0].reshape(batch,channel
,size,size)

        ############################################################
###
        #                          END OF YOUR CODE
#
        ############################################################
###


        return output
```

In [29]:
```
# define resnet building blocks

class ResidualBlock(nn.Module):
    def __init__(self, inchannel, outchannel, stride=1):

        super(ResidualBlock, self).__init__()

        self.left = nn.Sequential(Conv2D(inchannel, outchannel, ker
nel_size=3,
                                         stride=stride, padding=1,
bias=False),
                                  nn.BatchNorm2d(outchannel),
                                  nn.ReLU(inplace=True),
                                  Conv2D(outchannel, outchannel, ke
rnel_size=3,
                                         stride=1, padding=1, bias=
False),
                                  nn.BatchNorm2d(outchannel))

        self.shortcut = nn.Sequential()

        if stride != 1 or inchannel != outchannel:

            self.shortcut = nn.Sequential(Conv2D(inchannel, outchan
nel,
                                                 kernel_size=1, str
ide=stride,
                                                 padding = 0, bias=
False),
                                          nn.BatchNorm2d(outchannel
) )

    def forward(self, x):

        out = self.left(x)

        out += self.shortcut(x)

        out = F.relu(out)

        return out
```

In [30]:
```
# define resnet

class ResNet(nn.Module):

    def __init__(self, ResidualBlock, num_classes = 10):

        super(ResNet, self).__init__()

        self.inchannel = 64
        self.conv1 = nn.Sequential(Conv2D(3, 64, kernel_size = 3, s
```

```
tride = 1,
                                            padding = 1, bias = Fal
se),
                                  nn.BatchNorm2d(64),
                                  nn.ReLU())

        self.layer1 = self.make_layer(ResidualBlock, 64, 2, stride
= 1)
        self.layer2 = self.make_layer(ResidualBlock, 128, 2, stride
= 2)
        self.layer3 = self.make_layer(ResidualBlock, 256, 2, stride
= 2)
        self.layer4 = self.make_layer(ResidualBlock, 512, 2, stride
= 2)
        self.maxpool = MaxPool2D(4)
        self.fc = nn.Linear(512, num_classes)


    def make_layer(self, block, channels, num_blocks, stride):

        strides = [stride] + [1] * (num_blocks - 1)

        layers = []

        for stride in strides:

            layers.append(block(self.inchannel, channels, stride))

            self.inchannel = channels

        return nn.Sequential(*layers)


    def forward(self, x):

        x = self.conv1(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.maxpool(x)

        x = x.view(x.size(0), -1)

        x = self.fc(x)

        return x


def ResNet18():
    return ResNet(ResidualBlock)
```

Part 2 (40 points)

In this part, you will train the ResNet-18 defined in the previous part on the CIFAR-10 dataset. Code for loading the dataset, training and evaluation are provided.

Your Task

1. Train your network to achieve the best possible test set accuracy after a maximum of 10 epochs of training.
2. You can use techniques such as optimal hyper-parameter searching, data pre-processing
3. If necessary, you can also use another optimiser
4. **Answer the following question:** Given such a network with a large number of trainable parameters, and a training set of a large number of data, what do you think is the best strategy for hyperparameter searching?

YOUR ANSWER FOR 2.4 HERE

Answer: Bayesian Model-Based Optimization.

Each time when we use a different hyperparameters, the model need to be trained with data, make predictions on the validation data, and then use the validation algorithm. With a large number of hyperparameters and complex models such as deep neural networks this process can take lots of days. Bayesian Optimization process this process quicker than the previous method.

(also grid search and random search not as good as Bayesian Model-Based Optimization.)

In [31]:
```
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler

import torchvision.datasets as dset

import numpy as np

import torchvision.transforms as T


transform = T.ToTensor()


# load data

NUM_TRAIN = 49000
print_every = 100


data_dir = './data'
cifar10_train = dset.CIFAR10(data_dir, train=True, download=True, t
ransform=transform)
loader_train = DataLoader(cifar10_train, batch_size=64,
                          sampler=sampler.SubsetRandomSampler(range
(NUM_TRAIN)))

cifar10_val = dset.CIFAR10(data_dir, train=True, download=True, tra
nsform=transform)
loader_val = DataLoader(cifar10_val, batch_size=64,
                        sampler=sampler.SubsetRandomSampler(range(N
UM_TRAIN, 50000)))

cifar10_test = dset.CIFAR10(data_dir, train=False, download=True, t
ransform=transform)
loader_test = DataLoader(cifar10_test, batch_size=64)


USE_GPU = True
dtype = torch.float32

if USE_GPU and torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')
```

```
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
```

In [32]:
```
def check_accuracy(loader, model):
    # function for test accuracy on validation and test set
```

```
    if loader.dataset.train:
        print('Checking accuracy on validation set')
    else:
        print('Checking accuracy on test set')
    num_correct = 0
    num_samples = 0
    model.eval()  # set model to evaluation mode
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device
            y = y.to(device=device, dtype=torch.long)
            scores = model(x)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f)' % (num_correct, num_samp
les, 100 * acc))


def train_part(model, optimizer, epochs=1):
    """
    Train a model on CIFAR-10 using the PyTorch Module API.

    Inputs:
    - model: A PyTorch Module giving the model to train.
    - optimizer: An Optimizer object we will use to train the model
    - epochs: (Optional) A Python integer giving the number of epoc
hs to train for

    Returns: Nothing, but prints model accuracies during training.
    """
    model = model.to(device=device)  # move the model parameters to
CPU/GPU
    for e in range(epochs):
        print(len(loader_train))
        for t, (x, y) in enumerate(loader_train):
            model.train()  # put model to training mode
            x = x.to(device=device, dtype=dtype)  # move to device,
e.g. GPU
            y = y.to(device=device, dtype=torch.long)

            scores = model(x)
            loss = F.cross_entropy(scores, y)

            # Zero out all of the gradients for the variables which
the optimizer
            # will update.
            optimizer.zero_grad()

            loss.backward()
```

```
            # Update the parameters of the model using the gradient
s
            optimizer.step()

            if t % print_every == 0:
                print('Epoch: %d, Iteration %d, loss = %.4f' % (e,
t, loss.item()))
                #check_accuracy(loader_val, model)
                print()
```

In [ ]:

```
# code for optimising your network performance

##############################################################
#                         YOUR CODE HERE                      #
##############################################################



##############################################################
#                         END OF YOUR CODE                    #
##############################################################


# define and train the network
model = ResNet18()

# https://pytorch.org/docs/stable/optim.html
optimizer = optim.Adam(model.parameters(),lr=0.01, betas=(0.87, 0.1
), eps=1e-08, weight_decay=2, amsgrad=False)

train_part(model, optimizer, epochs = 10)


# report test set accuracy
check_accuracy(loader_test, model)


# save the model
torch.save(model.state_dict(), 'model.pt')
```

Part 3 (20 points)

The code provided below will allow you to visualise the feature maps computed by different layers of your network. Run the code (install matplotlib if necessary) and **answer the following questions**:

1. Compare the feature maps from low-level layers to high-level layers, what do you observe?
2. Use the training log, reported test set accuracy and the feature maps, analyse the performance of your network. If you think the performance is sufficiently good, explain why; if not, what might be the problem and how can you improve the performance?
3. What are the other possible ways to analyse the performance of your network?

YOUR ANSWER FOR PART 3 HERE

1. the image is getting more blurred and small.
2.
3. Drawing a diagram for error vs each iteration and analysis the performance of the model.

In [ ]:
```
#!pip install matplotlib

import matplotlib.pyplot as plt

plt.tight_layout()


activation = {}
def get_activation(name):
    def hook(model, input, output):
        activation[name] = output.detach()
    return hook

vis_labels = ['conv1', 'layer1', 'layer2', 'layer3', 'layer4']

for l in vis_labels:

    getattr(model, l).register_forward_hook(get_activation(l))


data, _ = cifar10_test[0]
data = data.unsqueeze_(0).to(device = device, dtype = dtype)

output = model(data)



for idx, l in enumerate(vis_labels):

    act = activation[l].squeeze()

    if idx < 2:
        ncols = 8
    else:
        ncols = 32

    nrows = act.size(0) // ncols

    fig, axarr = plt.subplots(nrows, ncols)
    fig.suptitle(l)


    for i in range(nrows):
        for j in range(ncols):
            axarr[i, j].imshow(act[i * nrows + j].cpu())
            axarr[i, j].axis('off')
```

=============== END OF CW2 ===============