RTML Lab 2: AlexNet vs. GoogLeNet

In this lab, we will develop PyTorch implementations of AlexNet and GoogleLeNet from scratch and compare them on CIFAR-10.

NAME = "Todsavad Tangtortan" ID = "123012"

Exercises

1. Create VSCode projects for each of these three networks. Be sure to properly define your Python classes, with one class per file and a main module that sets up your objects, runs the training process, and saves the necessary data.

2. Note that the AlexNet implementation here does not have the local response normalization feature described in the paper. Take a look at the PyTorch implementation of LRN and incorporate it into your AlexNet implementation as it is described in the paper. Compare your test set results with and without LRN.

3. Note that the backbone of the GoogLeNet implemented thus far does not correspond exactly to the description. Modify the architecture to

    A. Use the same backbone (input image size, convolutions, etc.) before the first Inception module
    B. Add the two side classifiers
4. Compare your GoogLeNet and AlexNet implementations on CIFAR-10. Comment on the number of parameters, speed of training, and accuracy of the two models on this dataset when trained from scratch.

5. Experiment with the pretrained GoogLeNet from the torchvision repository. Does it give better results on CIFAR-10 similar to what we found with AlexNet last week? Comment on what we can glean from the results about the capacity and generalization ability of these two models.

The report

Use the same format as last week. Describe your experiments and their results. The report should be turned in on Google Classroom before next week's lab.

In [ ]:
```
import torch
import torchvision
from torchvision import datasets, models, transforms
```

```
import torch.nn as nn
import torch.optim as optim
import time
import os
import copy
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import display
from PIL import Image
```

In [ ]:
```
from train_module import train_model
from test_module import test_model
```

2. AlexNet Implementation

- Note that the AlexNet implementation here does not have the local response normalization feature described in the paper.
- Take a look at the PyTorch implementation of LRN and incorporate it into your AlexNet implementation as it is described in the paper.

2.1 AlexNet using nn.Module with LRN

In [ ]:
```
class AlexNetModule_LRN(nn.Module):
    '''
    An AlexNet-like CNN

    Attributes
    ----------
    num_classes : int
        Number of classes in the final multinomial output layer
    features : Sequential
        The feature extraction portion of the network
    avgpool : AdaptiveAvgPool2d
        Convert the final feature layer to 6x6 feature maps by average pooling if t
    classifier : Sequential
        Classify the feature maps into num_classes classes
    '''
    def __init__(self, num_classes: int = 10) -> None:
        super().__init__()
        self.num_classes = num_classes
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.LocalResponseNorm(size=5,alpha=1e-4, beta=0.75, k=2), # added LRN
            nn.MaxPool2d(kernel_size=3, stride=2),

            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.LocalResponseNorm(size=5,alpha=1e-4, beta=0.75, k=2), # added LRN
            nn.MaxPool2d(kernel_size=3, stride=2),
```

```
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),

            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),

            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

2.2 AlexNet using nn.Module without LRN

In [ ]:

```
class AlexNetModule(nn.Module):
    '''
    An AlexNet-like CNN

    Attributes
    ----------
    num_classes : int
        Number of classes in the final multinomial output layer
    features : Sequential
        The feature extraction portion of the network
    avgpool : AdaptiveAvgPool2d
        Convert the final feature layer to 6x6 feature maps by average pooling if t
    classifier : Sequential
        Classify the feature maps into num_classes classes
    '''
    def __init__(self, num_classes: int = 10) -> None:
        super().__init__()
        self.num_classes = num_classes
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
```

```
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

- Compare your test set results with and without LRN.

In [ ]:
```
#Training Set
##AlexNet With LRN
'''
Epoch 9/9
----------
train Loss: 0.3815 Acc: 0.8657
Epoch time taken:  146.6789107322693
val Loss: 0.5868 Acc: 0.8014
Epoch time taken:  162.75838804244995
Training complete in 26m 2s
Best val Acc: 0.8014
'''
##AlexNet Withouts LRN
'''
Epoch 9/9
----------
train Loss: 0.4255 Acc: 0.8528
Epoch time taken:  132.84145975112915
val Loss: 0.6705 Acc: 0.7764
Epoch time taken:  149.2431561946869
Training complete in 25m 5s
Best val Acc: 0.7764
'''
```

In [ ]:
```
#Test Set
##AlexNet With LRN
'''
test Loss: 0.6008 Acc: 0.8029
Epoch time taken:  0.07869291305541992
'''
##AlexNet Withouts LRN
'''
test Loss: 0.6659 Acc: 0.7741
Epoch time taken:  0.08170104026794434
'''
```

3. GoogLeNet Implementation

- Note that the backbone of the GoogLeNet implemented thus far does not correspond exactly to the description.
- Modify the architecture to
    1. Use the same backbone (input image size, convolutions, etc.) before the first Inception module
    2. Add the two side classifiers

In [ ]:
```
import torch
import torch.nn as nn

class Flatten(nn.Module):
    def forward(self, x):
        batch_size = x.shape[0]
        return x.view(batch_size, -1)

class Inception(nn.Module):
    '''
    Inception block for a GoogLeNet-like CNN

    Attributes
    ----------
    in_planes : int
        Number of input feature maps
    n1x1 : int
        Number of direct 1x1 convolutions
    n3x3red : int
        Number of 1x1 reductions before the 3x3 convolutions
    n3x3 : int
        Number of 3x3 convolutions
    n5x5red : int
        Number of 1x1 reductions before the 5x5 convolutions
    n5x5 : int
        Number of 5x5 convolutions
    pool_planes : int
        Number of 1x1 convolutions after 3x3 max pooling
    b1 : Sequential
        First branch (direct 1x1 convolutions)
    b2 : Sequential
```

```
        Second branch (reduction then 3x3 convolutions)
    b3 : Sequential
        Third branch (reduction then 5x5 convolutions)
    b4 : Sequential
        Fourth branch (max pooling then reduction)
    '''

    def __init__(self, in_planes, n1x1, n3x3red, n3x3, n5x5red, n5x5, pool_planes):
        super(Inception, self).__init__()
        self.in_planes = in_planes
        self.n1x1 = n1x1
        self.n3x3red = n3x3red
        self.n3x3 = n3x3
        self.n5x5red = n5x5red
        self.n5x5 = n5x5
        self.pool_planes = pool_planes

        # 1x1 conv branch
        self.b1 = nn.Sequential(
            nn.Conv2d(in_planes, n1x1, kernel_size=1),
            nn.BatchNorm2d(n1x1),
            nn.ReLU(True),
        )

        # 1x1 conv -> 3x3 conv branch
        self.b2 = nn.Sequential(
            nn.Conv2d(in_planes, n3x3red, kernel_size=1),
            nn.BatchNorm2d(n3x3red),
            nn.ReLU(True),
            nn.Conv2d(n3x3red, n3x3, kernel_size=3, padding=1),
            nn.BatchNorm2d(n3x3),
            nn.ReLU(True),
        )

        # 1x1 conv -> 5x5 conv branch
        self.b3 = nn.Sequential(
            nn.Conv2d(in_planes, n5x5red, kernel_size=1),
            nn.BatchNorm2d(n5x5red),
            nn.ReLU(True),
            nn.Conv2d(n5x5red, n5x5, kernel_size=3, padding=1),
            nn.BatchNorm2d(n5x5),
            nn.ReLU(True),
            nn.Conv2d(n5x5, n5x5, kernel_size=3, padding=1),
            nn.BatchNorm2d(n5x5),
            nn.ReLU(True),
        )

        # 3x3 pool -> 1x1 conv branch
        self.b4 = nn.Sequential(
            nn.MaxPool2d(3, stride=1, padding=1),
            nn.Conv2d(in_planes, pool_planes, kernel_size=1),
            nn.BatchNorm2d(pool_planes),
            nn.ReLU(True),
        )

    def forward(self, x):
```

```
        y1 = self.b1(x)
        y2 = self.b2(x)
        y3 = self.b3(x)
        y4 = self.b4(x)
        return torch.cat([y1, y2, y3, y4], 1)
```

In [ ]:

```
import torch.nn as nn
from inception import Inception

class GoogLeNet(nn.Module):
    '''
    GoogLeNet-like CNN

    Attributes
    ----------
    pre_layers : Sequential
        Initial convolutional layer
    a3 : Inception
        First inception block
    b3 : Inception
        Second inception block
    maxpool : MaxPool2d
        Pooling layer after second inception block
    a4 : Inception
        Third inception block
    b4 : Inception
        Fourth inception block
    c4 : Inception
        Fifth inception block
    d4 : Inception
        Sixth inception block
    e4 : Inception
        Seventh inception block
    a5 : Inception
        Eighth inception block
    b5 : Inception
        Ninth inception block
    avgpool : AvgPool2d
        Average pool layer after final inception block
    linear : Linear
        Fully connected layer
    '''

    def __init__(self):
        super(GoogLeNet, self).__init__()
        self.pre_layers = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
            nn.ReLU(True),
            nn.MaxPool2d(3, stride=2, padding=1),
            nn.LocalResponseNorm(size=5,alpha=1e-4, beta=0.75, k=2),

            nn.Conv2d(64, 64, kernel_size=1),
            nn.ReLU(True),

            nn.Conv2d(64, 192, kernel_size=3,padding=1),
            nn.ReLU(True),
```

```
            nn.LocalResponseNorm(size=5,alpha=1e-4, beta=0.75, k=2)
        )

        self.a3 = Inception(192,  64,  96, 128, 16, 32, 32)
        self.b3 = Inception(256, 128, 128, 192, 32, 96, 64)

        self.maxpool = nn.MaxPool2d(3, stride=2, padding=1)

        self.a4 = Inception(480, 192,  96, 208, 16,  48,  64)
        self.aux1 = nn.Sequential(
            nn.AvgPool2d(5, stride=3),
            nn.Conv2d(512,128, kernel_size=1),
            nn.ReLU(True),
            Flatten(),
            nn.Linear(2048,1024),
            nn.ReLU(True),
            nn.Dropout(0.7),
            nn.Linear(1024,10)
        )

        self.b4 = Inception(512, 160, 112, 224, 24,  64,  64)
        self.c4 = Inception(512, 128, 128, 256, 24,  64,  64)
        self.d4 = Inception(512, 112, 144, 288, 32,  64,  64)
        self.aux2 = nn.Sequential(
            nn.AvgPool2d(5, stride=3),
            nn.Conv2d(528,128, kernel_size=1),
            nn.ReLU(True),
            Flatten(),
            nn.Linear(2048,1024),
            nn.ReLU(True),
            nn.Dropout(0.7),
            nn.Linear(1024,10)
        )
        self.e4 = Inception(528, 256, 160, 320, 32, 128, 128)

        self.a5 = Inception(832, 256, 160, 320, 32, 128, 128)
        self.b5 = Inception(832, 384, 192, 384, 48, 128, 128)

        self.avgpool = nn.AvgPool2d(8, stride=1)
        self.dropout = nn.Dropout(0.4)
        self.linear = nn.Linear(1024, 10)

    def forward(self, x):
        out = self.pre_layers(x)
        out = self.maxpool(out)

        out = self.a3(out)
        out = self.b3(out)

        out = self.maxpool(out)

        out = self.a4(out)
        aux_out1 = self.aux1(out)

        out = self.b4(out)
        out = self.c4(out)
```

```
        out = self.d4(out)
        aux_out2 = self.aux2(out)

        out = self.e4(out)
        out = self.maxpool(out)

        out = self.a5(out)
        out = self.b5(out)

        out = self.avgpool(out)
        out = self.dropout(out)

        out = out.view(out.size(0), -1)
        out = self.linear(out)

        return out, aux_out1, aux_out2
```

4. CIFAR-10 with GoogLeNet and AlexNet

- Compare your GoogLeNet and AlexNet implementations on CIFAR-10.

In [ ]:
```
#GoogleNet
#Training Set
'''
Epoch 9/9
----------
train Loss: 0.6107 Acc: 0.8964
Epoch time taken:  648.7454090118408
val Loss: 0.4725 Acc: 0.8507
Epoch time taken:  691.7330749034882
Training complete in 117m 23s
Best val Acc: 0.850700
'''
#Testing Set
'''
test Loss: 0.4734 Acc: 0.8501
Epoch time taken:  0.11796164512634277
'''
```

- Comment on the number of parameters, speed of training, and accuracy of the two models on this dataset when trained from scratch.

In [ ]:
```
# define a function for counting the number of parameters in a model
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```

In [ ]:
```
alexnet = AlexNetModule()
googlenet = GoogLeNet()

models = [alexnet, googlenet]
```

```
for model in models:
    print(f'{type(model).__name__} has {count_parameters(model):,} trainable parame
```

```
AlexNetModule has 57,044,810 trainable parameters
GoogLeNet has 10,635,134 trainable parameters
```

5. Pretrained GoogLeNet

- Experiment with the pretrained GoogLeNet from the torchvision repository.
- Does it give better results on CIFAR-10 similar to what we found with AlexNet last week?
- Comment on what we can glean from the results about the capacity and generalization ability of these two models.

In [ ]:
```
'''
googlenet_pre = torch.hub.load('pytorch/vision:v0.6.0', 'googlenet', pretrained=Tru
googlenet_pre.aux1.fc2 = nn.Linear(1024,10)
googlenet_pre.aux2.fc2 = nn.Linear(1024,10)
googlenet_pre.fc = nn.Linear(1024,10)
'''
```

In [ ]:
```
#AlexNet
#Training Set
'''
Epoch 9/9
----------
train Loss: 0.3163 Acc: 0.8951
Epoch time taken:  119.86784219741821
val Loss: 0.6379 Acc: 0.7931
Epoch time taken:  134.3369767665863
Training complete in 22m 39s
Best Val Acc: 0.7931
'''
#Testing Set
'''
test Loss: 0.5410 Acc: 0.8301
Epoch time taken:  0.09445977210998535
'''
#GoogLeNet
#Training Set
'''
Epoch 9/9
----------
train Loss: 0.2292 Acc: 0.9775
Epoch time taken:  499.1913321018219
val Loss: 0.2488 Acc: 0.9299
Epoch time taken:  531.9703154563904
Training complete in 91m 57s
Best val Acc: 0.929900
'''
#Testing Set
'''
test Loss: 0.2375 Acc: 0.9316
```

```
Epoch time taken:  0.08515691757202148
'''
```

Conclusion

Model	Train Loss	Train Acc	Val Loss	Val Acc	Test Loss	Test Acc	Time Taken
AlexNet with LRN	0.3815	86.57 %	0.5868	80.14 %	0.6008	80.29 %	26.2 m
AlexNet without LRN	0.4225	85.28 %	0.6705	77.64 %	0.6659	77.41 %	25.5 m
GoogLeNet	0.6107	89.64 %	0.4725	85.07 %	0.4734	85.01 %	117.23 m
pretrained AlexNet	0.3163	89.51 %	0.6379	79.31 %	0.5410	83.01 %	22.39 m
pretrained GoogLeNet	0.2292	97.75 %	0.2488	92.99 %	0.2375	93.16 %	91.57 m

- Due to different size, I has preprocessed as

  ```
  transforms.Resize(256),
  transforms.RandomCrop(224),
  transforms.RandomHorizontalFlip(),
  transforms.ToTensor(),
  transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
  0.225])
  ```
- For optimazation, I used SGD with the following configuration.

  ```
  lr=0.001
  momentum=0.9
  weight_decay=0.0005
  momentum=0.9
  ```
- I train each model for only 10 epochs, with batch size = 4.

- Section 1 : VsCode Project

  - This is my github to explore python file
  - https://github.com/guntsvzz/DSAI-AIT/tree/main/Course/Recent%20Trends%20in%20Machine%20Learning/Labolatory/02PyTorch-AlexNet-GoogLeNet
- Section 2 : AlexNet using local response normalization

  - LRN is inserted into convolutional layer 1 and 2 after ReLU, before MaxPooling.
  - The parameters of LRN are set according to the paper.
    `nn.LocalResponseNorm(size=5,alpha=1e-4, beta=0.75, k=2)`
  - Obviously, AlexNet with LRN outperforms than AlexNet without LRN
  - There is no significant difference a much between with and without LRN.
  - Models don't have indicate overfitting due to validation and test accuaracy with slightly near equally
- Section 3 : GoogLeNet

    - In the pre-layers part, adding 2 more convolutional layers together with MaxPooling and LRN.
    - Also adding the 2 auxillary branches which split out from a4 and d4 Inception module.
    - Due to adding auxillary branches, it requires 3 outputs to calculate the losses as well.
- Section 4 : CIFAR-10 with GoogLeNet and AlexNet
    - Even GoogLeNet will show the higher accuracy than AlexNet; however,
    - It has to trade off with time taken around 4-5 time of AlexNet
- Section 5 : Pretrained GoogLeNet
    - Due to pretrained GoogLeNet has trained with CIFAR-100 thus it required to change some output

      ```
      googlenet_pre = torch.hub.load('pytorch/vision:v0.6.0',
      'googlenet', pretrained=True, aux_logits = True)
      googlenet_pre.aux1.fc2 = nn.Linear(1024,10)
      googlenet_pre.aux2.fc2 = nn.Linear(1024,10)
      googlenet_pre.fc = nn.Linear(1024,10)
      ```
    - pretrained GoogLeNet give the highest accuracy comparing with any scratch versions. Both GoogLeNet achieve higher accuracy on CIFAR-10.