NATIONAL UNIVERSITY OF HO CHI MINH CITY
UNIVERSITY OF INFORMATION TECHNOLOGY
FACULTY OF INFORMATION SCIENCE AND ENGINEERING

COURSE PROJECT REPORT

IE229 - ARTIFICIAL INTELLIGENCE

TOPIC:

Evaluation of popular Deep Learning architectures for Image Classification problem on the CIFAR-10 dataset.

Lecturer: Takashi Ninomiya
Huỳnh Văn Tín
Lưu Thanh Sơn
Nguyễn Thành Luân

Student group:

1. Nguyễn Văn Khoa ID: 18520929
2. Lê Thị Thu Hằng ID: 18520274
3. Nguyễn Thị Hồng Nhung ID: 18521218

Ho Chi Minh City, 7/2021

Artificial Intelligence Final Report Assignment 問題1 (Problem 1)

レポート解答用紙 (Report Answer Sheet)

Group Leader

学生証番号 (Student ID): 18520929

名前(Name): Nguyễn Văn Khoa

Group Members

学生証番号 (Student ID): 18520274

名前(Name): Lê Thị Thu Hằng

学生証番号 (Student ID): 18521218

名前(Name): Nguyễn Thị Hồng Nhung

問題1 (Problem 1)のレポート

LECTURE'S COMMENTS

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Preface

Computer vision (CV) is the field of Computer Science that focuses on replicating parts of the complexity of the human vision system and enabling computers to identify and process objects in images and videos in the same way that humans do. Until recently, CV only worked in limited capacity. Thanks to advances in Artificial Intelligence (AI) and innovations in Deep Learning, the field has been able to take great leaps in recent years and has been able to surpass humans in some tasks related to detecting and labeling objects.

Along with a tremendous amount of visual data (more than 3 billion images are shared online every day), the computing power required to analyze the data is now accessible. As the field of CV has grown with new hardware and algorithms so has the accuracy rates for object identification. In less than a decade, today's systems have reached 99 percent accuracy from 50 percent making them more accurate than humans at quickly reacting to visual inputs. Early experiments in CV started in the 1950s and it was first put to use commercially to distinguish between typed and handwritten text by the 1970s, today the applications for computer vision have grown exponentially.

There are many problems in the field of CV, some of the notable problems being *Image Classification, Object Detection, Image Segmentation, Image Generation, Image Captioning, Multi-Object Tracking* and so on.

Image classification basically just involves labelling an image based on the content of the image. There would generally be a fixed set of labels and the model will have to predict the label that best fits the image. This problem is hard for a machine as, all it sees is just a stream of numbers in an image. *Object detection* is also a CV problem in which, given an image or video frame, the algorithm must decide the locations of one or more target objects, this problem involves recognizing various sub images and drawing a bounding box around each recognized sub image. This is slightly more complicated to solve as compared to classification problem and the

algorithm must play around with the image co-ordinates a lot more. *Image segmentation* is a problem in which algorithms not only understand the locations of objects in images or video frames, but to map the boundaries more precisely between different objects in the same image.

In this report, we will use Python programming language and Python libraries for Data Science research (e.g. NumPy, Pandas, PyTorch) to install experimentally popular Deep Learning architecture for Image Classification problem on the CIFAR-10 dataset. For easy tracking and clarification of information, the report will be divided into the following sections:

- **Section 1**: Overview.
- **Section 2**: Dataset.
- **Section 3**: Baseline Models.
- **Section 4**: VGG16 & VGG19 Architectures.
- **Section 5:** Resnet50 & Resnet152 Architectures.
- **Section 5**: Summary of results.
- **Section 6:** Conclusion.

Section 1: OVERVIEW

1.1 Problem Description

As mentioned above, Image Classification is one of the main problems in Computer Vision that attempts to connect an image to a set of class labels. It usually is a supervised learning problem, wherein a set of pre-labeled training data is fed to a machine learning algorithm. The algorithm attempts to ***learn the visual features*** contained in the training images associated with each label and classify unlabeled images accordingly.

Up to now, there have been many methods from many approaches to solve this problem, typically as traditional machine learning and deep learning.

Traditional machine learning uses hand-crafted features

In traditional machine learning approach, humans spend a good amount of time in manual features selection and engineering. This process relies on human's domain knowledge (or experts) to create features which make machine learning algorithms can distinguish objects. Some of the handcrafted feature sets are:

- Histogram of Oriented Gradients (HOG)
- Haar Cascades
- Scale-Invariant Feature Transform (SIFT)
- Speeded Up Robust Feature (SURF)

Then the produced features will be feed into a classifier like Support Vector Machines (SVM) or Adaboost to predict the output.

Deep learning automatically extracts features

In deep learning, humans do not need to manually extract features from the image. The network automatically extracts features and learns their importance on the output by applying weights to its connections. The raw image will be feed into to the network and, as it passes through the network layers, it identifies patterns within the

image to create features. Neural networks can be thought of as feature extractors + classifiers which are end-to-end trainable as opposed to traditional ML models that use hand-crafted features.

As can be seen, the most obvious difference between the two approaches is the input of the classification algorithm. One side needs to have the human feature extraction process, the other side does it automatically. Indeed, Feature Extraction is a core component of the CV pipeline. In fact, the entire deep learning model works around the idea of extracting **useful features** which clearly define the objects in the image. A feature is a measurable piece of data in the dataset which is unique to this specific object when compared with others. It may be a distinct color in an image or a specific shape such as a line, edge, or an image segment. Selecting good features that clearly distinguish objects and increases the predictive power of classifier. Practice has shown that deep learning models which automatically extracts features achieve higher accuracy than humans, because deep learning can learn features that humans cannot even see and interpret.

To actualize the above thing, methods coming from the Deep Learning approach will use some Convolutional Neural Network (CNN) architectures such as VGG, Resnet in which convolution layers, pooling layers and fully connected layers as its core.

There are many datasets to get started with the Image Classification problem. The most famous one is the MNIST Handwritten Digits dataset, which has been used in the early age of CV and is still used as an introduction to Image Classification problems. Although this dataset has played an essential role in the development of CV, the task is considered trivial for the current state of the field and industry. As a result, more challenging datasets have been created for research purposes such as STL-10, CIFAR-10, and CIFAR-100. After all, CIFAR-10 is also the dataset used in this work.

1.2 Challenges, targets

1.2.1 Challenges

- **In term of methodology**: The image classification problem has many methods from many different approaches. Therefore, the comparison, evaluation and selection of appropriate methods to improve accuracy is a matter of concern and attention.
- **In term of data**: There are many difficulties when looking at the data for this problem, typically can be mentioned as *Intra-Class Variation, Scale Variation, View-Point Variation, Occlusion, Illumination, Background Clutter.*

1.2.2 Targets

In this report, we will figure out and implement baseline model, besides, we will also build different Deep Learning architectures that apply CNN [1] to better extract features (e.g. VGG16, VGG19, Resnet).

To achieve that, the following targets have been set:

(1) – Be able to implement baseline model.

(2) – Be able to apply Deep Learning network architectures using CNN in feature extraction to improve accuracy.

(3) – Compare, analyze, and evaluate the accuracy of the above methods.

Section 2. DATASET

In this Section, we will present a detailed description of the CIFAR-10 dataset. This dataset consists of 60,000 colored (RGB) images of dimensions 32×32 pixels divided into 10 classes (6,000 images per class). The dataset is divided into five training batches and one test batch, each with 10,000 images. The data review process is as follows:

First, we will import necessary libraries with the following code:

```
1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4
5 from torchvision import datasets, transforms
6 from torch.utils.data import DataLoader
7 from sklearn.metrics import confusion_matrix
8
9 import matplotlib.pyplot as plt
10 import numpy as np
```

Figure 1:

Before we download the CIFAR10 dataset, we need to setup to convert images from the dataset to PyTorch tensors. The way to do that is to create a variable transform and apply the function transforms.ToTensor().

```
1 transform = transforms.ToTensor()
```

Figure 2:

To automatically download the CIFAR10 dataset we will create a data_path variable and use a simple string (these two dots mean that we are going one level up from the current directory) to specify the path. Next, we create an object cifar10_train and from datasets, we call the function CIFAR10(). As arguments to this function, we

will provide data_path, train which is set to True because this part of the data set will be training set. The third argument is a download which is also set to True. Then, with the same function, we will create an object cifar10_test where testing data will be stored. The only difference is that here, we will set the argument train to False. This means that this part of the data set will be test set.

```
1 data_path = '../data_cifar/'
2 cifar10_train = datasets.CIFAR10(data_path, train=True, download=True, transform=transform)
3 cifar10_test = datasets.CIFAR10(data_path, train=False, download=True, transform=transform)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ../data_cifar/cifar-10-python.tar.gz
170499072/? [00:02<00:00, 57962308.85it/s]

Extracting ../data_cifar/cifar-10-python.tar.gz to ../data_cifar/
Files already downloaded and verified
```

Figure 3:

Now, if we check our training and testing data, we can see that variable cifar10_train consists of 50,000 images and variable cifar10_test consists of 10,000 images.

```
1 print("Training: ", len(cifar10_train))
2 print("Testing: ", len(cifar10_test))
```

```
Training:  50000
Testing:  10000
```

Figure 4:

If we check the type of our training data, we can see that it is a very specific torchvision data type. Basically, this torchvision.datasets.cifar.CIFAR10 is a collection of tensors organized together.

```
1 print(type(cifar10_train))
2 print(type(cifar10_test))
```

```
<class 'torchvision.datasets.cifar.CIFAR10'>
<class 'torchvision.datasets.cifar.CIFAR10'>
```

Figure 5:

By using indexing, we can get back a tensor as an output. What is interesting here is that this is a tuple of two items. The first item is a 32×32 pixel image, and the second item is the label. Now we unpack this tuple and check the type and shape of one image.

```
1 type(cifar10_train[0])
```

```
tuple
```

```
1 image, label = cifar10_train[0]
2 type(image)
```

```
torch.Tensor
```

```
1 image.shape
```

```
torch.Size([3, 32, 32])
```

Figure 6:

We can see that image is a tensor of 3×32×32 where number 3 represents three channels of red, green, and blue color and 32×32 pixels is the size of the image.

Also, if we print classes and the label. We can see that the value of the label is 6. If we take a look at the class names, we will see that it is a frog.

```
classes = cifar10_train.classes
print (classes)
print(label)
print(classes[label])
```

```
['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
6
frog
```

Figure 7:

Next, we will consider the number of each class in the training set by using the following script:

```
import pandas as pd
class_count = {}
for _, index in cifar10_train:
  label = classes[index]
  if label not in class_count:
    class_count[label] = 0
  class_count[label] += 1
class_count_df = pd.Series(class_count)
class_count_df
```

```
frog          5000
truck         5000
deer          5000
automobile    5000
bird          5000
horse         5000
ship          5000
cat           5000
dog           5000
airplane      5000
dtype: int64
```

Figure 8:

It can be seen that the number of data points per class is equal, which will not cause problems in terms of data imbalance.

Now, let's visualize this image from our dataset. Note that order of the dimensions is 3×32×32. However, to plot the image with Matplotlib the order should

be 32×32×3. So, to change the order of the dimensions we will use the function permute().

```
1 plt.imshow(image.permute(1, 2, 0))
```

```
<matplotlib.image.AxesImage at 0x7f5e73a6b450>
```

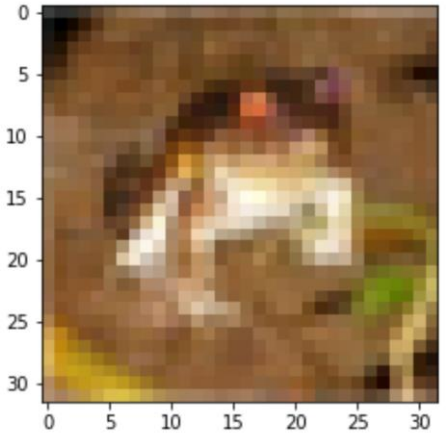

Figure 9:

It is important to understand that to build the Neural Network we will work with many parameters. For this reason, it makes sense to load training data in batches using the DataLoader class. What we are going to do is to use the following code:

```
1 torch.manual_seed(77)
2 train_loader = DataLoader(cifar10_train, batch_size=64, shuffle=True)
3 test_loader = DataLoader(cifar10_test, batch_size=64, shuffle=False)
```

Figure 10:

Here, we are using the function manual_seed() which sets the seed for generating random numbers. Then we are going to create train_loader and test_loader objects using the function DataLoader(). For the train_loader object as parameters, we will pass our training data cifar10_train and batch_size that will automatically group images into batches. For this example, we will use a batch size of 64. The last parameter is shuffle which will shuffle our data after we pass the boolean value of

True. Then, we create a test_loader using the same code. The only difference is that we don't really need to shuffle the data. That is why the shuffle parameter will be set to False. Also, a batch_size for a test_loader will be equal to 64.

The last thing, let's consider more images in this dataset as follows:

```
from torchvision.utils import make_grid
for images, _ in train_loader:
    plt.figure(figsize=(16,8))
    plt.axis('off')
    plt.imshow(make_grid(images, nrow=16).permute((1, 2, 0)))
    break
```

Figure 11:

Figure 12:

Section 3. BASELINE MODELS

3.1 Baseline implementation

First, we followed the 10th lecture to implement a simple fully connected neural network structure as follows:

```python
class Baseline(torch.nn.Module):
  def __init__(self):
    super(Baseline, self).__init__()
    self.l1 = torch.nn.Linear(3072, 300)
    self.l2 = torch.nn.Linear(300, 300)
    self.l3 = torch.nn.Linear(300, 10)

  def forward(self,x):
    h = F.relu(self.l1(x))
    h = F.relu(self.l2(h))
    y = self.l3(h)
    return y
```

```
Baseline(
  (l1): Linear(in_features=3072, out_features=300, bias=True)
  (l2): Linear(in_features=300, out_features=300, bias=True)
  (l3): Linear(in_features=300, out_features=10, bias=True)
)
```

Figure 13:

This network consists of 3 layers. The first layer has an input size of (3072, 300) with 3072 corresponding to 32x32x3 being the size of the image. Next, the second layer (300x300) will be a hidden layer fully connected between the 300 nodes of the input layer and its 300 nodes. The final layer of size (300x10) will connect 300 nodes of the second layer with 10 nodes representing 10 classes of the problem.

We have also defined helper functions to serve the training and testing process. A part of the source code will be briefly described in the image below.

```python
def train_baseline(version = 1, EPOCH=10):
  train_acc_lst = []
  test_acc_lst = []
  loss_lst = []
  test_loss_lst = []

  model = Baseline().to(DEVICE)
  print(model)
  optimizer = torch.optim.Adam(model.parameters())
  for epoch in range(EPOCH):
    model.train()
    loss = 0
    train_correct = 0
    train_total = 0
    for images, labels in train_loader:
      images = images.view(-1, 32*32*3).to(DEVICE)
      labels = labels.to(DEVICE)
      optimizer.zero_grad()
      y = model(images)
      batchloss = F.cross_entropy(y,labels)
      batchloss.backward()
      optimizer.step()
      loss = loss + batchloss.item()
```

```python
def test():
  correct = 0
  total = len(test_loader.dataset)
  model = Baseline().to(DEVICE)
  model.load_state_dict(torch.load(MODEL_NAME))
  model.eval()
  for images, labels in test_loader:
    images = images.view(-1, 32*32*3).to(DEVICE)
    labels = labels.to(DEVICE)
    y = model(images)
    pred_labels = y.max(dim=1)[1]
    correct = correct + (pred_labels == labels).sum()

  print("correct: ", correct.item())
  print("total: ", total)
  print("accuracy: ", correct.item()/total)
```

Figure 14:

In addition, we have also defined some functions to visualize the training parameters in the form of histograms.

```python
def visualize_acc(train_acc, test_acc, name, title):
  x = [i for i in range(10)]
  y1 = [x for x in train_acc]
  y2 = [x for x in test_acc]

  plt.plot(x, y1, label='Train accuracy')
  plt.plot(x, y2, label='Test accuracy')
  plt.xticks([i for i in range(10)])
  plt.xlabel("Epoch")
  plt.ylabel("Accuracy")
  plt.title(title)

  plt.legend()
  plt.savefig(name)
  plt.show()
```

```python
def visualize_loss(train_loss, test_loss, name, title):
  x = [i for i in range(10)]
  y1 = [x for x in train_loss]
  y2 = [x for x in test_loss]

  plt.plot(x, y1, label='Train loss')
  plt.plot(x, y2, label='Test loss')
  plt.xticks([i for i in range(10)])
  plt.xlabel("Epoch")
  plt.ylabel("Loss")
  plt.title(title)

  plt.legend()
  plt.savefig(name)
  plt.show()
```

Figure 15:

After training with 10 epochs, the obtained parameters are as follows:

```
======================================================================
epoch 0: loss = 1428.0037825107574, Train_acc = 0.33728, Test_acc = 0.4038
======================================================================
epoch 1: loss = 1286.9604938030243, Train_acc = 0.41228, Test_acc = 0.4433
======================================================================
epoch 2: loss = 1219.3808616399765, Train_acc = 0.44506, Test_acc = 0.4303
======================================================================
epoch 3: loss = 1178.8636728525162, Train_acc = 0.46204, Test_acc = 0.4629
======================================================================
epoch 4: loss = 1150.4643050432205, Train_acc = 0.47494, Test_acc = 0.475
======================================================================
epoch 5: loss = 1121.389346241951, Train_acc = 0.48824, Test_acc = 0.4835
======================================================================
epoch 6: loss = 1100.780465245247, Train_acc = 0.4975, Test_acc = 0.4928
======================================================================
epoch 7: loss = 1079.8915742635727, Train_acc = 0.50722, Test_acc = 0.4821
======================================================================
epoch 8: loss = 1060.832926094532, Train_acc = 0.51554, Test_acc = 0.4997
======================================================================
epoch 9: loss = 1043.9866364002228, Train_acc = 0.52074, Test_acc = 0.4991
```

Figure 16:

Section 4. VGG16 & VGG19 ARCHITECTURES

4.1 Theoretical basis

VGG [2] is a classical Convolutional Neural Network architecture proposed by Karen Simonyan and Andrew Zisserman from the University of Oxford in 2014. This model participated in ILSVRC (ImageNet Large Scale Visual Recognition Competition) 2014 and achieved excellent results: ranked second in the classification task and first in the localization task.

VGG has smaller convolution filters (3x3) with more depth, and its simplicity characterizes it: the only other components are pooling layers and fully connected layers.

All configurations follow the general design in architecture and just differences from the depth of 11 weight-layers (VGG11) to 19 weight-layer (VVG19). The width of convolution layers is small, starting from 64 in the first and increasing by a factor of 2 after each max-pooling layer until it reaches 51. In both variations of VGG Network, there consist of 3 Fully Connected layers, and the last fully connected layer uses softmax activation function for classification purposes.

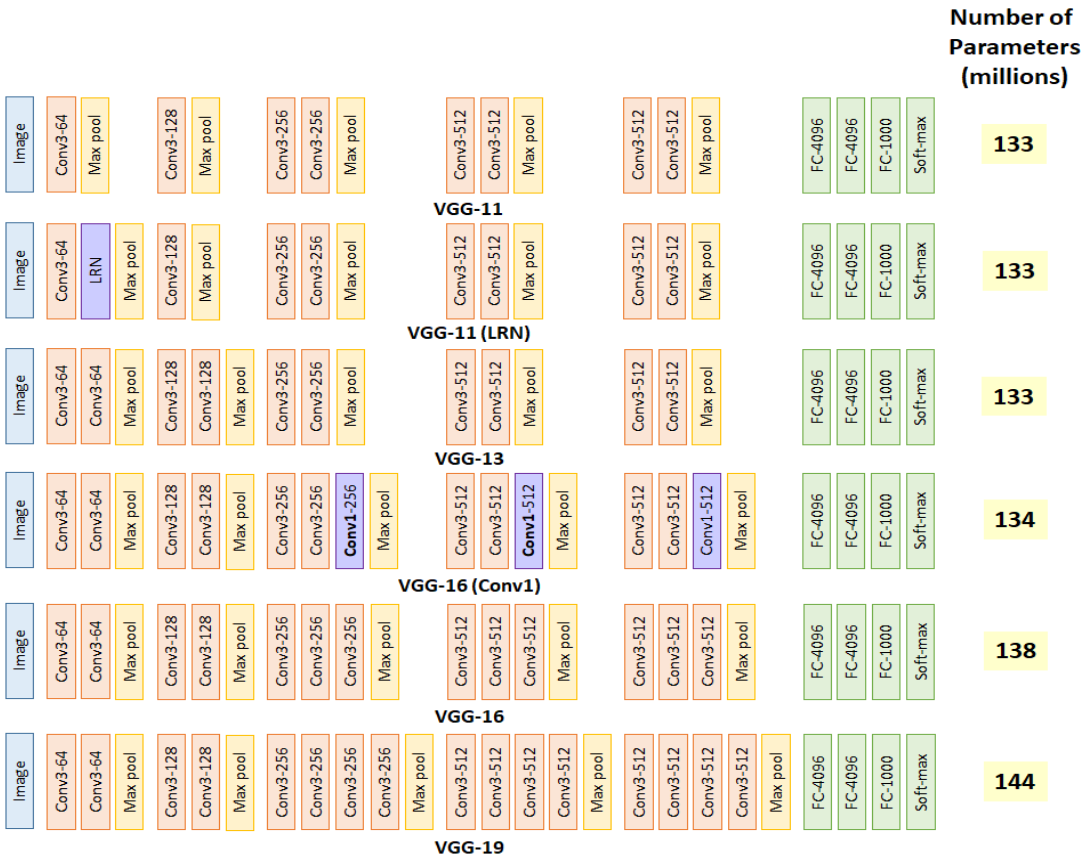

Figure 17:

Although the architect of VGG Network is simple, the number of parameters is huge. These parameters include the weights of the Convolution kernel and the Fully Connected layer. The VGG network has many parameters that make its use highly efficient, but it also has obvious disadvantages:

- It takes a long time to train the model, so the adjustment is difficult.
- Large storage capacity, not conducive to deployment in embedded systems.

4.2 Implementation

4.2.1 *VGG16 architecture*

The following are specifics architectures of the VGG16:

- 13 Convolutional layers: denoted by conv3-ABC (ABC represents the number of channels of the Convolutional layer).
- 3 Fully Connected layers: denoted by FC-ABCD (ABCD represents the number of channels of the Fully Connected layer).
- 5 Pooling: denoted by max-pooling.

The convolution layer (13 layers) and the fully connected layer (3 layers) have weighting coefficients, so they are considered as weight layers. The total number of layers is 16, so it is called VGG16. The pooling layer has no weight, so it is not counted.

```
class VGG16(nn.Module):
  def __init__(self):
    super(VGG16, self).__init__()
    self.conv1_1 = nn.Conv2d(3,64,kernel_size=3,padding=1)
    self.conv1_2 = nn.Conv2d(64,64,kernel_size=3,padding=1)
    self.batchnorm1 = nn.BatchNorm2d(64, track_running_stats=False)

    self.conv2_1 = nn.Conv2d(64,128,kernel_size=3,padding=1)
    self.conv2_2 = nn.Conv2d(128,128,kernel_size=3,padding=1)
    self.batchnorm2 = nn.BatchNorm2d(128, track_running_stats=False)

    self.conv3_1 = nn.Conv2d(128,256,kernel_size=3, padding=1)
    self.conv3_2 = nn.Conv2d(256,256,kernel_size=3, padding=1)
    self.conv3_3 = nn.Conv2d(256,256,kernel_size=3, padding=1)
    self.batchnorm3 = nn.BatchNorm2d(256, track_running_stats=False)

    self.conv4_1 = nn.Conv2d(256,512,kernel_size=3,padding=1)
    self.conv4_2 = nn.Conv2d(512,512,kernel_size=3,padding=1)
    self.conv4_3 = nn.Conv2d(512,512,kernel_size=3,padding=1)
    self.batchnorm4 = nn.BatchNorm2d(512, track_running_stats=False)

    self.conv5_1 = nn.Conv2d(512,512,kernel_size=3,padding=1)
    self.conv5_2 = nn.Conv2d(512,512,kernel_size=3,padding=1)
    self.conv5_3 = nn.Conv2d(512,512,kernel_size=3,padding=1)
    self.batchnorm5 = nn.BatchNorm2d(512, track_running_stats=False)

    self.maxpool = nn.MaxPool2d(2,2)

    self.fc1 = nn.Linear(1*1*512,4096)
    self.fc2 = nn.Linear(4096,4096)
    self.fc3 = nn.Linear(4096,10)
```

13 Convolutional layers + Batchnorm

Maxpool layer
(5 maxpool layers are the same)

3 Fully Connected layers

Figure 18:

In this work, we **apply extra Batch norm layers** behind convolutional blocks to achieve a better effect. The primary purpose of batch normalization is to normalize the data in batch layers to a normal distribution so that gradient descent converges much faster and improves accuracy.

```
def forward(self, x, training=True):
  x = F.relu(self.batchnorm1(self.conv1_1(x)))
  x = F.relu(self.batchnorm1(self.conv1_2(x)))
  x = self.maxpool(x)

  x = F.relu(self.batchnorm2(self.conv2_1(x)))
  x = F.relu(self.batchnorm2(self.conv2_2(x)))
  x = self.maxpool(x)

  x = F.relu(self.batchnorm3(self.conv3_1(x)))
  x = F.relu(self.batchnorm3(self.conv3_2(x)))
  x = F.relu(self.batchnorm3(self.conv3_3(x)))
  x = self.maxpool(x)

  x = F.relu(self.batchnorm4(self.conv4_1(x)))
  x = F.relu(self.batchnorm4(self.conv4_2(x)))
  x = F.relu(self.batchnorm4(self.conv4_3(x)))
  x = self.maxpool(x)

  x = F.relu(self.batchnorm5(self.conv5_1(x)))
  x = F.relu(self.batchnorm5(self.conv5_2(x)))
  x = F.relu(self.batchnorm5(self.conv5_3(x)))
  x = self.maxpool(x)

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

  x = F.relu(self.fc1(x))
  x = F.dropout(x, 0.5, training=training)
  x = F.relu(self.fc2(x))
  x = F.dropout(x, 0.5, training=training)
  return self.fc3(x)
```

Figure 19:

4.2.2 *VGG19 architecture*

Specific analysis of VGG19 includes:

- 16 Convolutional layers: denoted by conv3-ABC (ABC represents the number of channels of the Convolutional layer).
- 3 Fully Connected layers: denoted by FC-ABCD (ABCD represents the number of channels of the Fully Connected layer).
- 5 Pooling: denoted by max-pooling.

Figure 20: Init layers in VGG19 model

Like the VGG16 model, we also use the Batchnorm layer behind each Convolutional block to increase model performance.

```
def forward(self, x, training=True):
  x = F.relu(self.batchnorm1(self.conv1_1(x)))
  x = F.relu(self.batchnorm1(self.conv1_2(x)))
  x = self.maxpool(x)

  x = F.relu(self.batchnorm2(self.conv2_1(x)))
  x = F.relu(self.batchnorm2(self.conv2_2(x)))
  x = self.maxpool(x)

  x = F.relu(self.batchnorm3(self.conv3_1(x)))
  x = F.relu(self.batchnorm3(self.conv3_2(x)))
  x = F.relu(self.batchnorm3(self.conv3_3(x)))
  x = F.relu(self.batchnorm3(self.conv3_4(x)))
  x = self.maxpool(x)

  x = F.relu(self.batchnorm4(self.conv4_1(x)))
  x = F.relu(self.batchnorm4(self.conv4_2(x)))
  x = F.relu(self.batchnorm4(self.conv4_3(x)))
  x = F.relu(self.batchnorm4(self.conv4_4(x)))
  x = self.maxpool(x)

  x = F.relu(self.batchnorm5(self.conv5_1(x)))
  x = F.relu(self.batchnorm5(self.conv5_2(x)))
  x = F.relu(self.batchnorm5(self.conv5_3(x)))
  x = F.relu(self.batchnorm5(self.conv5_4(x)))
  x = self.maxpool(x)

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

  x = F.relu(self.fc1(x))
  x = F.dropout(x, 0.5, training=training)
  x = F.relu(self.fc2(x))
  x = F.dropout(x, 0.5, training=training)
  return self.fc3(x)
```

Figure 21: Forward function in VGG19 model

Section 5. RESNET50 & RESNET152 ARCHITECTURES

5.1 Theoretical basis

Kaiming He et al. published ResNet (Residual Neural Network) [3]. ResNet has shortcut connections in every residual block to address the degradation problem. The problem is caused by adding more layers to a suitably deep model leads to higher training error. Figure below shows the example of a shortcut connection in a residual block:

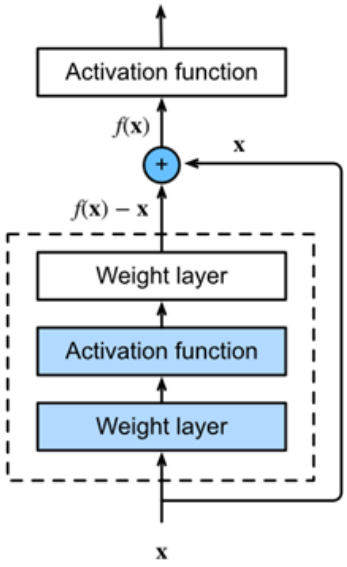

Figure 22: Residual block.

We denote the input by x and assume that the desired underlying mapping we want to obtain by learning is f(x), to be used as the input to the activation function on the top. The portion within the dotted-line box needs to learn the residual mapping f(x)−x, which is how the residual block derives its name. If the identity mapping f(x)=x is the desired underlying mapping, the residual mapping is easier to learn: we only need to push the weights and biases of the upper weight layer (e.g., fully-connected layer and convolutional layer) within the dotted-line box to zero. The solid line carrying the layer input x to the addition operator is called a shortcut connection.

With residual blocks, inputs can forward propagate faster through the residual connections across layers [4].

5.2 Implementation

In our experiments, we train the networks using the Adam optimizer by minimizing the binary cross-entropy, epoch = 10 and batch_size = 64. We use 50 – layer ResNet and 152 – layer ResNet with the architecture shown in the Table 1.

Table 1: ResNet model architectures that we have implemented.

Layer name	50 – layer	152 – layer
conv1	$3\ x\ 3, 64$	
	$3\ x\ 3$ max pool, stride 2	
conv2_x	$\begin{bmatrix} 1\ x\ 1, & 64 \\ 3\ x\ 3, & 64 \\ 1\ x\ 1, & 256 \end{bmatrix} x\ 3$	$\begin{bmatrix} 1\ x\ 1, & 64 \\ 3\ x\ 3, & 64 \\ 1\ x\ 1, & 256 \end{bmatrix} x\ 3$
conv3_x	$\begin{bmatrix} 1\ x\ 1, & 128 \\ 3\ x\ 3, & 128 \\ 1\ x\ 1, & 512 \end{bmatrix} x\ 4$	$\begin{bmatrix} 1\ x\ 1, & 128 \\ 3\ x\ 3, & 128 \\ 1\ x\ 1, & 512 \end{bmatrix} x\ 8$
conv4_x	$\begin{bmatrix} 1\ x\ 1, & 256 \\ 3\ x\ 3, & 256 \\ 1\ x\ 1, & 1024 \end{bmatrix} x\ 6$	$\begin{bmatrix} 1\ x\ 1, & 256 \\ 3\ x\ 3, & 256 \\ 1\ x\ 1, & 1024 \end{bmatrix} x\ 36$
conv5_x	$\begin{bmatrix} 1\ x\ 1, & 512 \\ 3\ x\ 3, & 512 \\ 1\ x\ 1, & 2048 \end{bmatrix} x\ 3$	$\begin{bmatrix} 1\ x\ 1, & 512 \\ 3\ x\ 3, & 512 \\ 1\ x\ 1, & 2048 \end{bmatrix} x\ 3$
	Average pool, fully connected	

```
class Block(nn.Module):
  def __init__(self, input_channels, output_channels, stride=1):
    super(Block, self).__init__()
    self.block = nn.Sequential(
        nn.Conv2d(input_channels, output_channels,
                  kernel_size=1, bias = False),
        nn.BatchNorm2d(output_channels),
        nn.ReLU(inplace=True),
        nn.Conv2d(output_channels, output_channels, stride=stride,
                  kernel_size=3, padding=1, bias=False),
        nn.BatchNorm2d(output_channels),
        nn.ReLU(inplace=True),
        nn.Conv2d(output_channels, output_channels*4,
                  kernel_size = 1, bias=False),
        nn.BatchNorm2d(output_channels*4),
    )

    self.shortcut = nn.Sequential()

    if stride != 1 or input_channels != output_channels * 4:
      self.shortcut = nn.Sequential(
          nn.Conv2d(input_channels, output_channels * 4, stride = stride,
                    kernel_size = 1, bias = False),
                    nn.BatchNorm2d(output_channels *4)
      )
```

Figure 23: Class of a residual block.

```
class ResNet(nn.Module):
  def __init__(self, block, num_block):
    super(ResNet, self).__init__()
    self.in_channels = 64

    self.conv1 = nn.Sequential(
        nn.Conv2d(3,64, kernel_size = 3, padding =1, bias=False),
        nn.BatchNorm2d(64),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
    )
    self.conv2_x = self._make_layer(block, 64, num_block[0])
    self.conv3_x = self._make_layer(block, 128, num_block[1], 2)
    self.conv4_x = self._make_layer(block, 256, num_block[2], 2)
    self.conv5_x = self._make_layer(block, 512, num_block[3], 2)

    self.pool = nn.AdaptiveAvgPool2d((1,1))

    self.fc = nn.Linear(512*4, 10)
```

Figure 24: General structure of the ResNet model.

```
model = ResNet(Block, [3,4,6,3]).to(DEVICE)
optimizer = torch.optim.Adam(model.parameters())
```

Figure 25: 50 – layer ResNet model initial.

```
model = ResNet(Block, [3, 8, 36, 3]).to(DEVICE)
optimizer = torch.optim.Adam(model.parameters())
```

Figure 26: 152 – layer ResNet model initial.

Section 6. SUMMARY OF RESULTS

6.1 Training charts

The following are some graphs that visualize the change of parameters during the training process.

Baseline model:

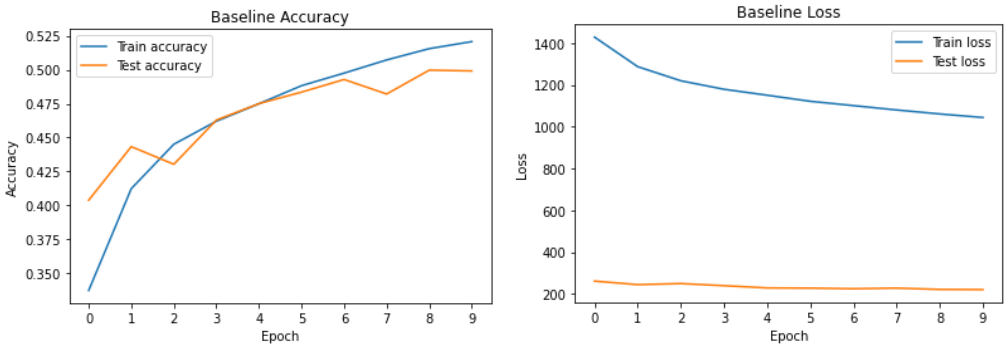

Figure 27:

VGG16:

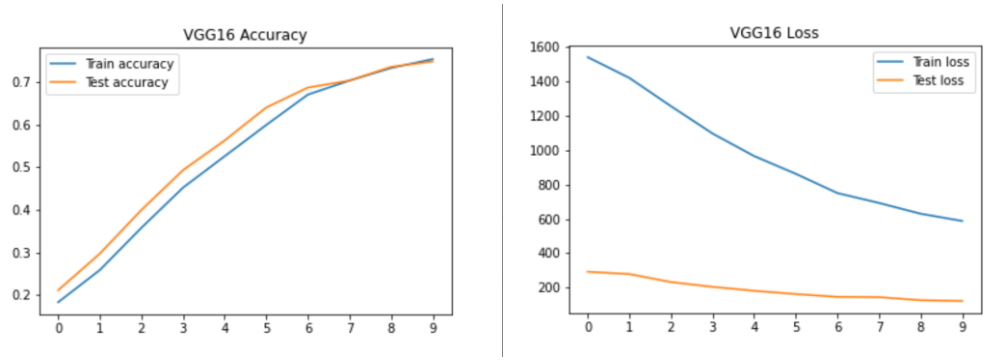

Figure 28:

VGG19:

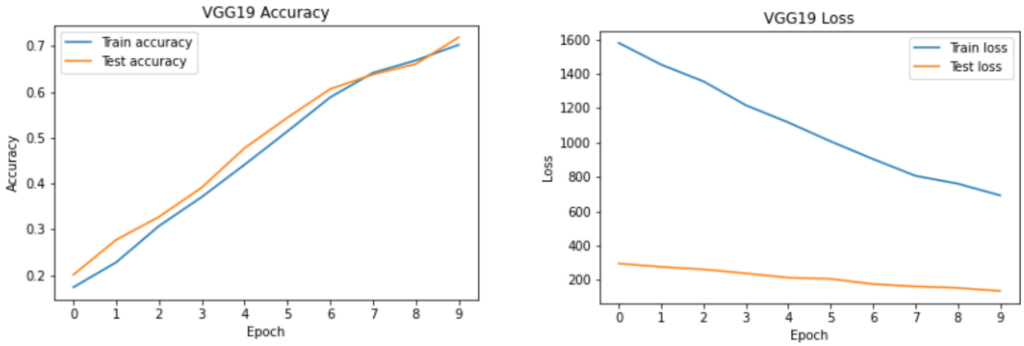

Figure 29:

ResNet50:

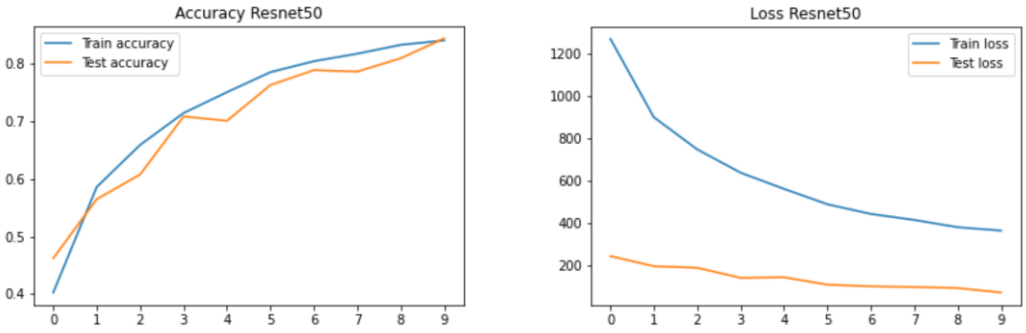

Figure 30:

ResNet152:

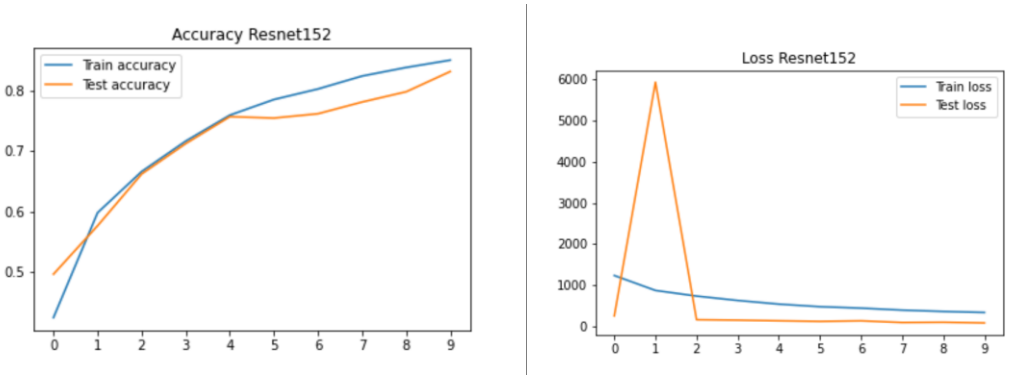

Figure 31:

6.2 Sumary of results

The results of the models are summarized in the following table 2.

Table 2: The results of the models.

Model	**Accuracy**
Baseline	0.4991
VGG16	0.7655
VGG19	0.7165
ResNet50	**0.8329**
ResNet152	0.8229

It can be seen that the ResNet50 method has the highest accuracy with an accuracy of 0.8329. Followed by ResNet152 (0.8229). In general, accuracy from ResNet is greater than 0.8, for VGG methods it is 0.7 and baseline model is only 0.5.

CONCLUSION

We achieved our target of building models for this task of reaching over 75% on accuracy evaluation metric. ResNet152 model achieve 0.8229. ResNet50 model achieve 0.8329 and is the best model. In future, we aim to apply some typical hyperparameters turning technique for solving this task.

REFERENCES

[1] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, 'Flexible, high performance convolutional neural networks for image classification', 2011.

[2] K. Simonyan and A. Zisserman, 'Very deep convolutional networks for large-scale image recognition', *arXiv preprint arXiv:1409.1556,* 2014.

[3] K. He, X. Zhang, S. Ren, and J. Sun, 'Deep residual learning for image recognition', in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[4] J. Quinn, J. McEachen, M. Fullan, M. Gardner, and M. Drummy, *Dive into deep learning: Tools for engagement*. Corwin Press, 2019.