**Artificial Intelligence Final Report Assignment 問題1 (Problem 1)**

**レポート解答用紙 (Report Answer Sheet)**

**(Group 35)**

Group Leader

学生証番号 (Student ID): 21522357

名前(Name): Lê Hải Nam

Group Members

学生証番号 (Student ID): 21522429

名前(Name): Trần Nguyễn Yến Nhi

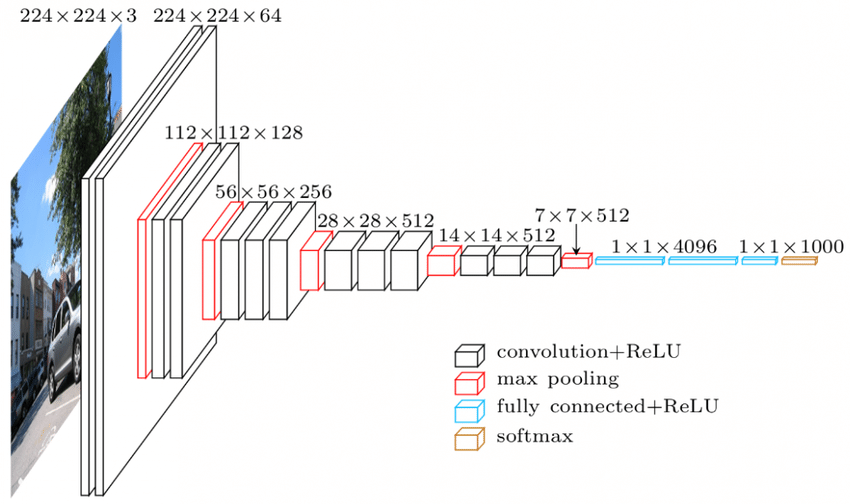
学生証番号 (Student ID): 21522315

名前(Name): Nguyễn Thị Cẩm Ly

問題1 (Problem 1)のレポート

## **Idea:**

To improve the version of the program in the 10th lecture (or the program was written in Lab Work (4)). We use the VGG16 model (CNN model) instead of the Conv2d layer - method that brings the best results in the 10th lecture (with accuracy is about 61%).



*Figure 1: VGG16 model.*

**The VGG-16 model** is a convolutional neural network (CNN) architecture that was proposed by the Visual Geometry Group (VGG) at the University of Oxford. It is characterized by its depth, consisting of 16 layers, including 13 convolutional layers and 3 fully connected layers. The model’s architecture features a stack of convolutional layers followed by max-pooling layers, with progressively increasing depth. This design enables the model to learn intricate hierarchical representations of visual features, leading to robust and accurate predictions.

## **Program:**

1. **Prepare**

Prepare images for training and testing: resizes train images to 50x50 pixels, applies random horizontal flips for augmentation, and converts them to tensors. While test images: converts images to tensors and resizes them to 50x50 pixels for consistency in input dimensions across training and testing phases.

# Define transformations for training and testing data

transform\_train = tv.transforms.Compose([

# Resize images to 50x50 pixels

tv.transforms.Resize((50,50)),

# Randomly flip images horizontally

tv.transforms.RandomHorizontalFlip(),

# Convert images to PyTorch tensors

tv.transforms.ToTensor(),

])

transform\_test = tv.transforms.Compose([

# Convert images to PyTorch tensors

tv.transforms.ToTensor(),

# Resize images to 50x50 pixels

tv.transforms.Resize((50,50)),

])

1. **Define model**

The VGG class implementation defines a convolutional neural network architecture, specifically tailored to VGG16, featuring alternating convolutional and pooling layers followed by a fully connected classifier for image classification tasks. Its use of batch normalization and ReLU activation ensures efficient training and the forward method orchestrates feature extraction and classification seamlessly, encapsulating a robust deep learning model suitable for various computer vision applications.

VGG16 = [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M']

class VGG(nn.Module):

def \_\_init\_\_(self, vgg\_name):

super(VGG, self).\_\_init\_\_()

self.features = self.\_make\_layers(VGG16)

self.classifier = nn.Linear(512, 10)

def forward(self, x):

out = self.features(x)

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

out = self.classifier(out)

return out

def \_make\_layers(self, cfg):

layers = []

in\_channels = 3

for x in VGG16:

if x == 'M':

layers += [nn.MaxPool2d(kernel\_size=2, stride=2)]

else:

layers += [nn.Conv2d(in\_channels, x, kernel\_size=3, padding=1),

nn.BatchNorm2d(x),

nn.ReLU(inplace=True)]

in\_channels = x

layers += [nn.AvgPool2d(kernel\_size=1, stride=1)]

return nn.Sequential(\*layers)

1. **Train model**

DEVICE = "cuda" if torch.cuda.is\_available() else "cpu"

def train():

chart\_x = []

chart\_y = []

optimizer = torch.optim.Adam(model.parameters())

for epoch in range(EPOCH):

loss = 0

for images, labels in train\_loader:

images = images.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()

print("epoch", epoch, ": loss", loss)

chart\_x.append(epoch)

chart\_y.append(loss)

plt.style.use('seaborn-whitegrid')

plt.xlabel("epoch")

plt.ylabel("loss")

plt.plot(chart\_x,chart\_y, color = "red")

1. **Test model**

def test():

total = len(test\_loader.dataset)

correct = 0

# Set the model to evaluation mode

model.eval()

for images, labels in test\_loader:

images = images.to(DEVICE)

labels = labels.to(DEVICE)

y = model(images)

pred\_labels = y.max(dim=1)[1]

correct= correct + (pred\_labels == labels).sum()

'''Computes total correct predictions (correct).

Calculates accuracy (accuracy = correct / total) and prints it as a percentage.

Outputs the number of correct predictions and total number of samples for verification.'''

print("correct: ", correct.item())

print("total: ", total)

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

**Execution Results:**

Using VGG16 significantly enhances accuracy, achieving a peak accuracy of 89.34% for our model.

1. **Program:**

EPOCH = 10

print('\n' + 'total EPOCH:', EPOCH)

#EPOCHS = [10, 20, 50] List of epochs to iterate over

EPOCH = 10

print('\n' + 'total EPOCH:', EPOCH)

model = VGG('VGG16').to(DEVICE)

train()

test()

EPOCH = 20

print('\n' + 'total EPOCH:', EPOCH)

model = VGG('VGG16').to(DEVICE)

train()

test()

EPOCH = 50

print('\n' + 'total EPOCH:', EPOCH)

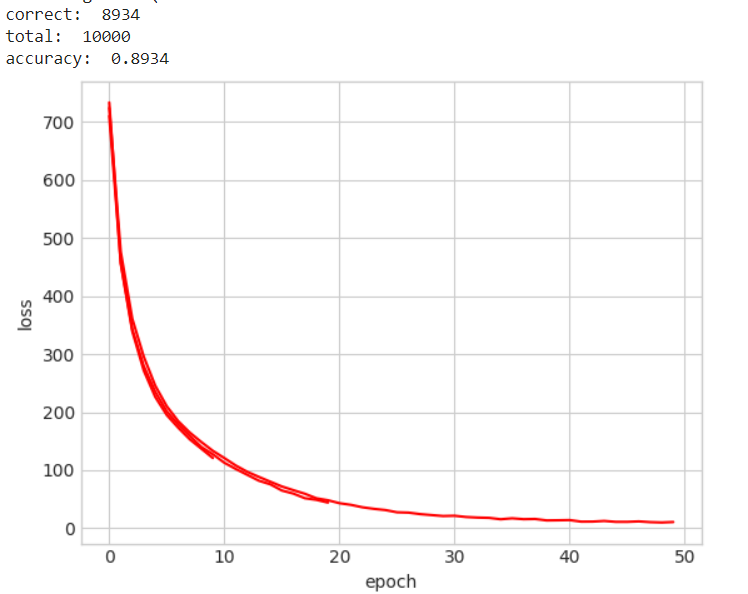
model = VGG('VGG16').to(DEVICE)

train()

test()

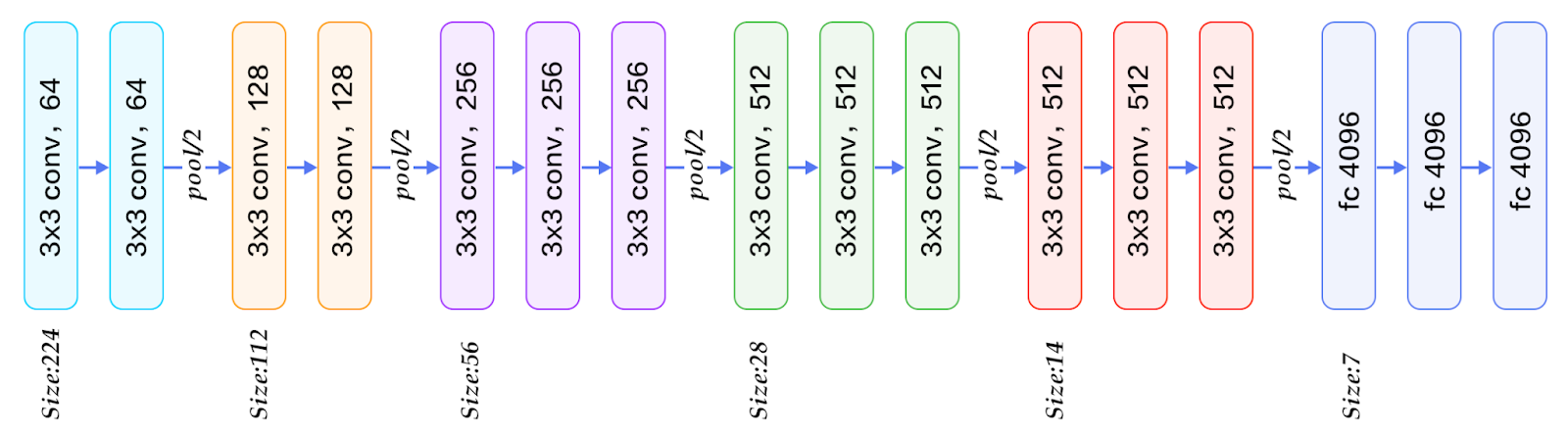
1. **Result:**

Highest accuracy is **89,34%** at 50 epoch.

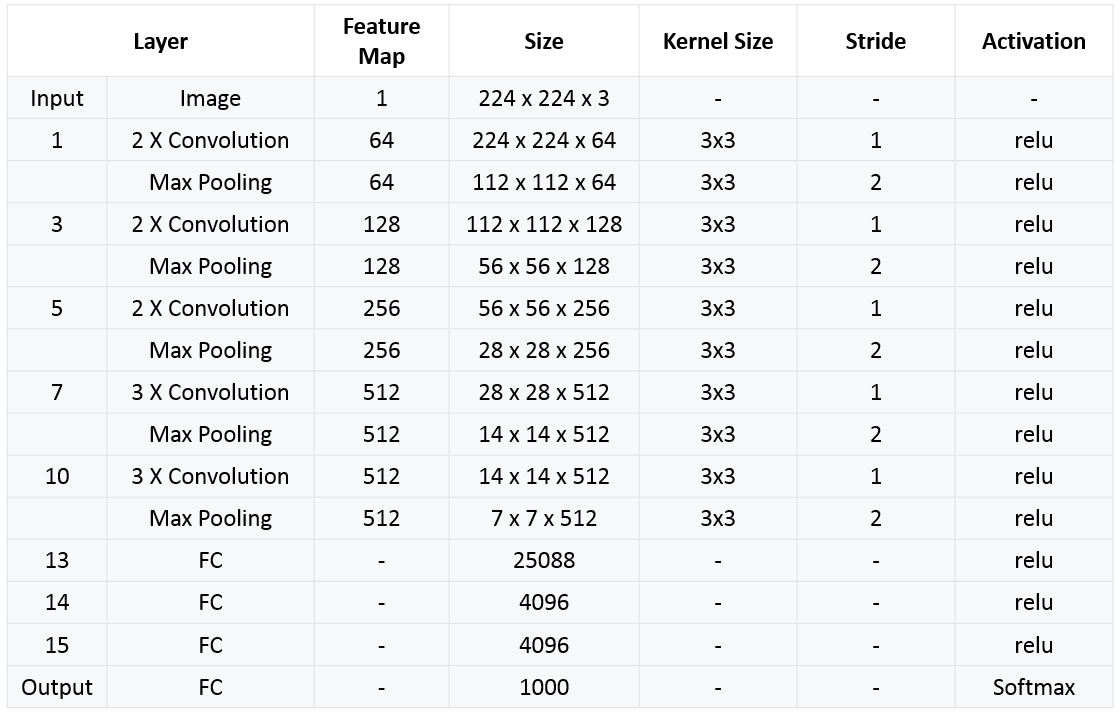
  
*Figure 2: Line plot of Mean Squared Error Loss over Training 50 Epoch*

## **Explanation:**

The input of VGG is set to an RGB image of 224x244 size. The average RGB value is calculated for all images on the training set image, and then the image is input to the VGG convolutional network. A 3x3 or 1x1 filter is used, and the convolution step is fixed. . There are 3 VGG fully connected layers, which can vary from VGG11 to VGG19 according to the total number of convolutional layers + fully connected layers. The minimum VGG11 has 8 convolutional layers and 3 fully connected layers. The maximum VGG19 has 16 convolutional layers. +3 fully connected layers. In addition, the VGG network is not followed by a pooling layer behind each convolutional layer, or a total of 5 pooling layers distributed under different convolutional layers. The following figure is VGG Structure diagram:



*Figure 3: VGG-16 Architecture Map*



*Figure 4:Table shows how VGG-16 works.*

VGG-16 contains 16 layers and VGG19 contains 19 layers. A series of VGGs are exactly the same in the last three fully connected layers. The overall structure includes 5 sets of convolutional layers, followed by a MaxPool. The difference is that more and more cascaded convolutional layers are included in the five sets of convolutional layers.

Using the VGG16 model typically yields better results compared to a simple model with only one Conv2d layer for several key reasons related to the architecture and capabilities of the neural network. Here’s a detailed explanation:

1. **Number of Layers and Model Depth**

VGG16: VGG16 is a deep neural network with 16 layers, including multiple convolutional (Conv2d), batch normalization (BatchNorm2d), and pooling (MaxPool2d) layers. Each convolutional layer helps the network learn complex features from the input data. The depth of the model allows VGG16 to learn features at various levels, from simple edges and corners to complex shapes and structures.

Simple Conv2d Model: A model with only one Conv2d layer has very limited capability to learn features from the input data. The lack of depth and variety of layers means the model cannot capture complex features of images.

1. **Consecutive Convolutional Layers**

VGG16: Uses many convolutional layers, combined with pooling layers to gradually reduce the spatial dimensions of the input. Consecutive convolutional layers allow the model to learn features in small regions of the image and then combine them to capture global information.

Simple Conv2d Model: Uses a single Conv2d layer only has one layer. This limits the model's ability to capture complex features and lacks the capability to learn features at multiple levels.

1. **Feature Learning Capability**

VGG16: With many convolutional and nonlinear (ReLU) layers, VGG16 has a superior ability to learn nonlinear features. Nonlinear layers enable the model to learn complex relationships within the data.

Simple Conv2d Model: Has fewer layers and lacks many nonlinear layers, hence its ability to learn nonlinear features is very limited.

1. **Pooling and Normalization Layers**

VGG16: Uses MaxPooling layers to reduce the spatial dimensions of the input, helping to reduce the number of parameters and computations, while also creating a hierarchical structure of learned features. Additionally, the use of normalization layers (BatchNorm) helps the model stabilize and train faster.

Simple Conv2d Model: Does not include pooling and normalization layers, making the model harder to train and less stable.

⇒ The VGG16 model features a more complex and optimized design, with multiple consecutive convolutional layers, pooling, and normalization techniques that allow it to learn features from simple to complex. This results in a stable and efficient training process. On the other hand, a model with only one Conv2d layer lacks depth and the ability to learn complex features, leading to poorer performance.