CNN Models on Natural Images

April 29, 2019

Aaron Skipper, Yu Xi

Introduction

Classification problems are types of problems that present itself in many professional industries. Experts and professionals within their perspective fields use classification techniques to better operate and understand problems that their businesses are faced with, and these types issues are general enough to explain several problems across a general working industry. We are motivated with the idea of continuing to enhance our learning and understanding of classification problems, and we selected the Kaggle produced natural image dataset to experiment with.

We will attempt to train a convolution neural network (CNN) to correctly classify these images according their classes. We utilized the PyTorch framework to express our experimenting within the CNN. For image analysis, CNN works well as they specialize in making sense of patterns. We will experiment with the analysis of the problem statements of how accurately we can build a neural network model that classifies the 8 classes within the dataset and identify which parameters within the model influence the neural network's accuracy and time to perform.

Description of the data set

Data was downloaded from Kaggle and consists of 6899 images. The images are all made up of 8 classes: person, motorbike, fruit, flower, car, airplane, cat, dog.

Description of the deep learning network and training algorithm

We choose to use to use CNN to address this classification problem on the nature dataset. Our baseline model consists of 2 convolutional layers using the framework PyTorch. We used batch size of 128 images, with each image having a dimension of 224x224. Each layer of the network has parameter settings that will influence the layer interactions along with the performance of the network. The first convolution layer has input nodes set to 16, and output nodes set at 32. The convolutional filters (kernels) are set to be 3x3. The batch normalization setting that follow the convolutions layer, is used to address the network phenomena of internal covariate shift. Mean and std of that layer is considered and can be used as tuning parameters for the layer, with batch normalization, but they were not used in our experimenting. Covariate shift is when inputs of a layer consistently change during training, due to the other layers' input activations changes that occur within those layers. ReLU, non-linear activation functions are then used following the convolution setting. The max pooling method of convolution is applied to each layer, which acts to reduce computational load as out dimensions are decreased after applied. In the final layers of the network, the fully connected layer is necessary to complete the pattern recognition process and perform the classifications. This ensures that all the nodes at the end of the network will be used for classification. The SoftMax transfer function was used as the output layer, which serves well for pattern recognition networks with multiple neurons.

Experimental setup

Firstly, we randomly split the data set into three parts: 80% training, 10% validation, 10% testing. The training dataset was used to fit the model. The validation dataset was used to provide an unbiased evaluation of the model, as well as tuning the hyperparameters. A large difference between classification accuracies of training and validation dataset indicates an overfitting problem. The test dataset was used to evaluate the final model we choose.

After data partitioning, image transformations were performed on the training set to augment the data. Transformations including random resized crop, random rotation, color jitter, and random horizontal flip were used. Then, all the images were normalized to the same size. For example, Figure 1-1 shows a random dog image from the dataset, and Figure 1-2 shows 16 possible ways it could be randomly transformed.

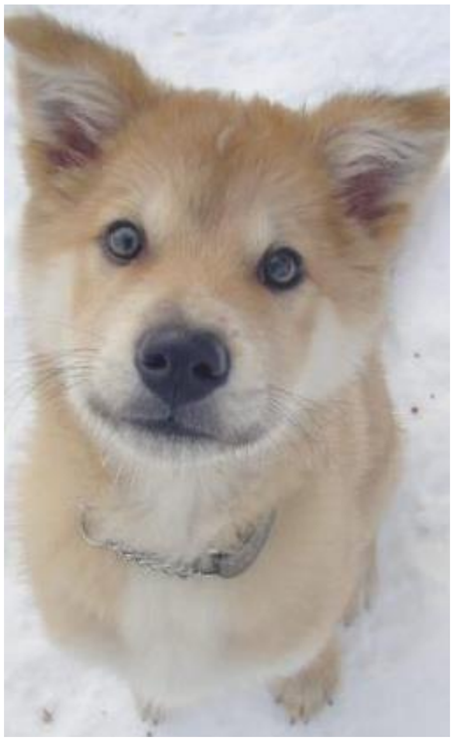


Figure 1-1. Example dog image

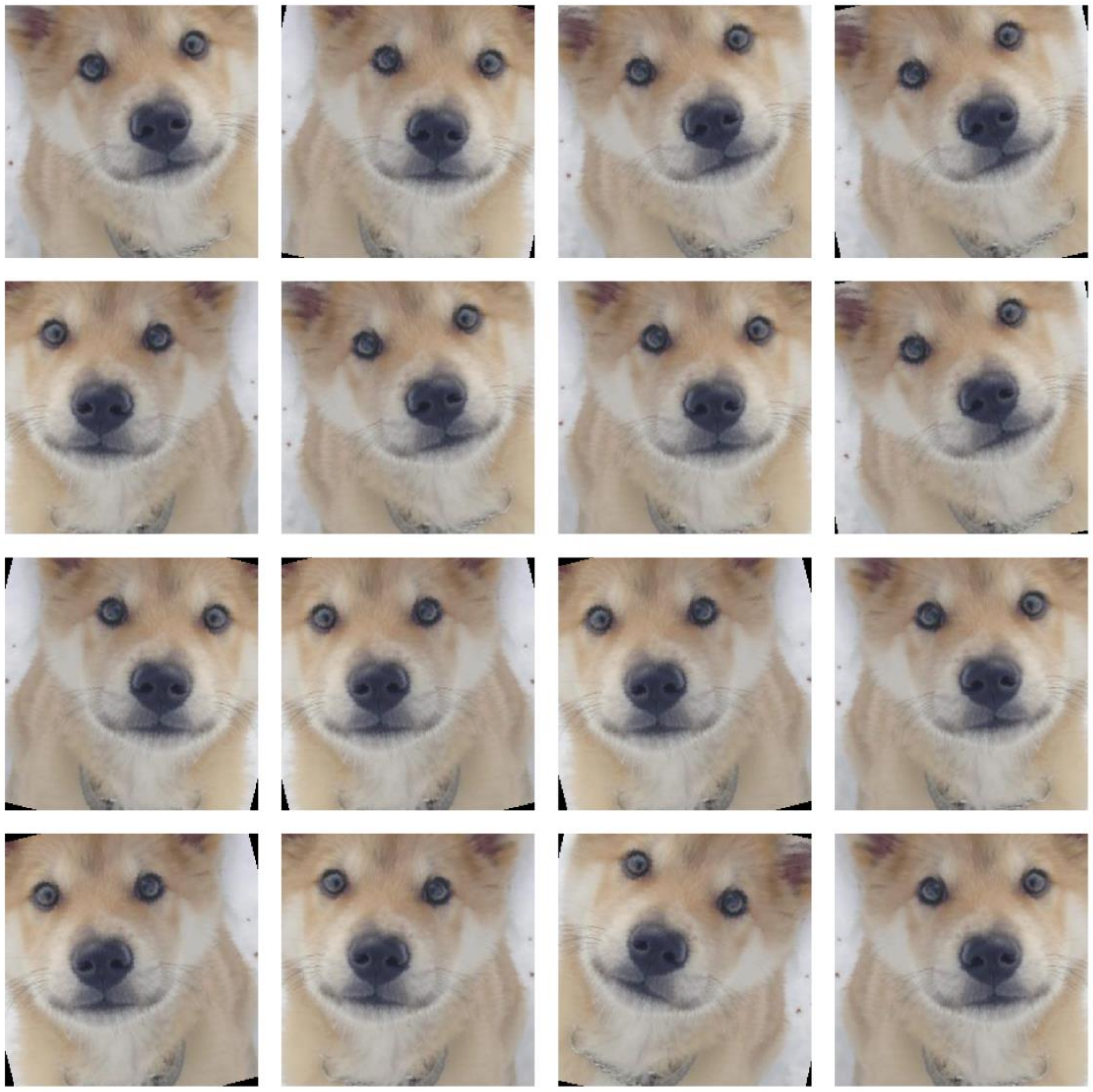


Figure 1-2. Example dog images after random transformation

For the hyperparameters, we fixed some of them because their combinations are too many to tune. Since VGG-16 is one of the famous CNN models for image classification, we use it as a reference to choose some of the hyperparameters, including batch size: 128, transfer function: ReLU, kernel size: (3, 3), stride: 2 and padding: (1, 1). Since this is a classification problem, we choose cross entropy as the loss function and Adam as the optimizer based on our prior experience. The hyperparameters left for us to tune are the learning rate, dropout rate, and the number of epochs.

Models will be experimented from 2 to 6 layers, and the hyperparameters will be changed manually according to the loss and accuracies on training and validation dataset for each experiment. We will first determine the learning rate while keeping the model and other parameters fixed. Once we choose the best learning rate, we will use it for all models assuming it performs similarly while changing other parameters. If the accuracy on the training dataset is low, we will increase the number of epochs. If there is a large difference between accuracies of training and validation dataset, we will decrease the number of epochs to prevent overfitting. Once we choose the best model, we will introduce the dropout rate to the model and find the best one.

Results

The first model we had experiments on is a 2-layer CNN. The architecture of the model is shown in Figure 2. The numbers besides the layers are the dimensions of the corresponding layers. We keep the number of epochs at 5 and vary the learning rate. The accuracy of model is shown in Table 1. From the three models, learning rate with 1e-03 gives the best performance on the validation dataset. We will use this learning rate for the rest of our models.

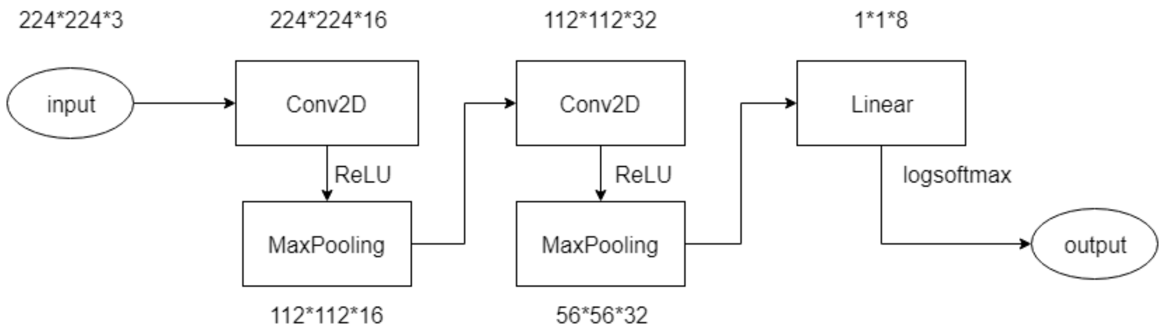
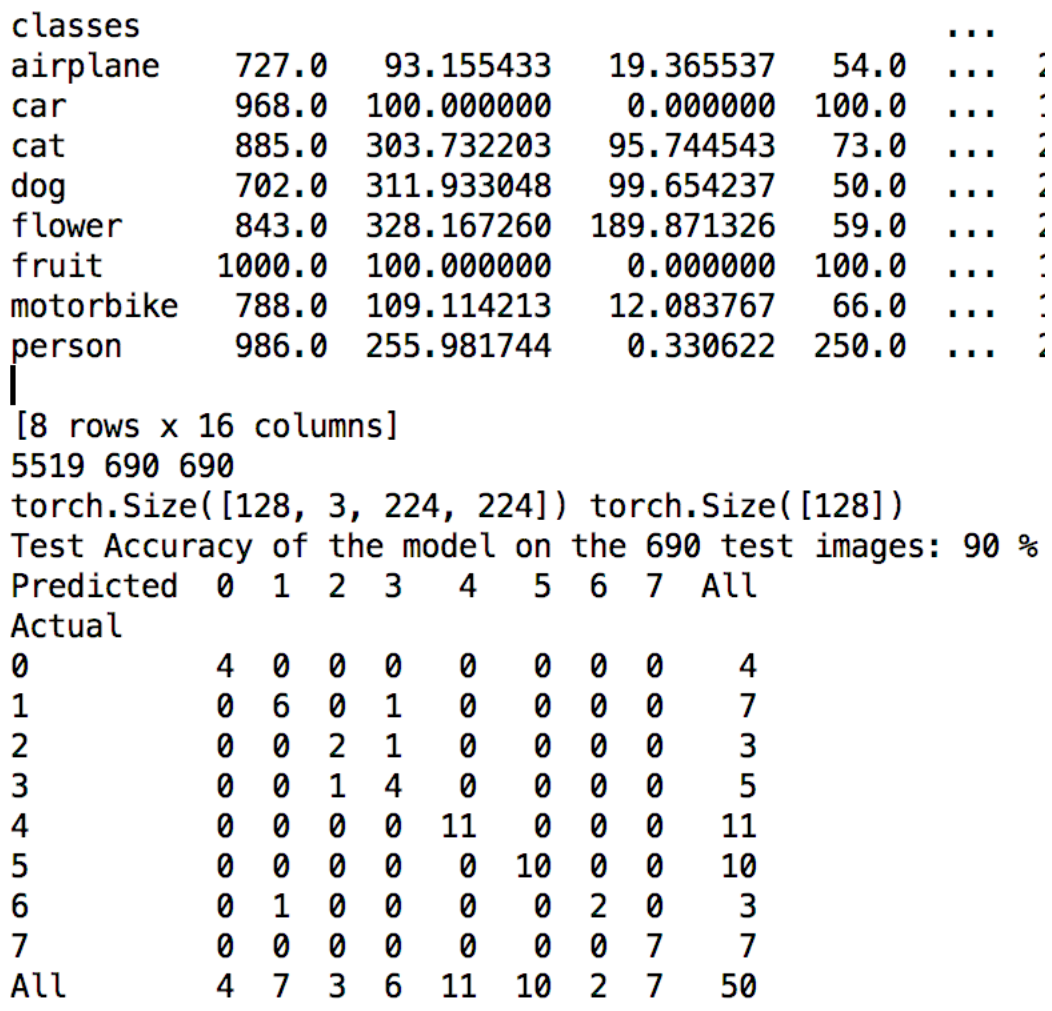


Figure 2. Architecture of a 2-layer CNN

Table 1. Accuracy of a 2-layer CNN on training and validation dataset

Learning Rate	Training Accuracy (%)	Validation Accuracy (%)
1e-02	91.4	83.8
1e-03	98.6	90.2
1e-04	98.4	66.7

Confusion matrix for 2-layer CNN:



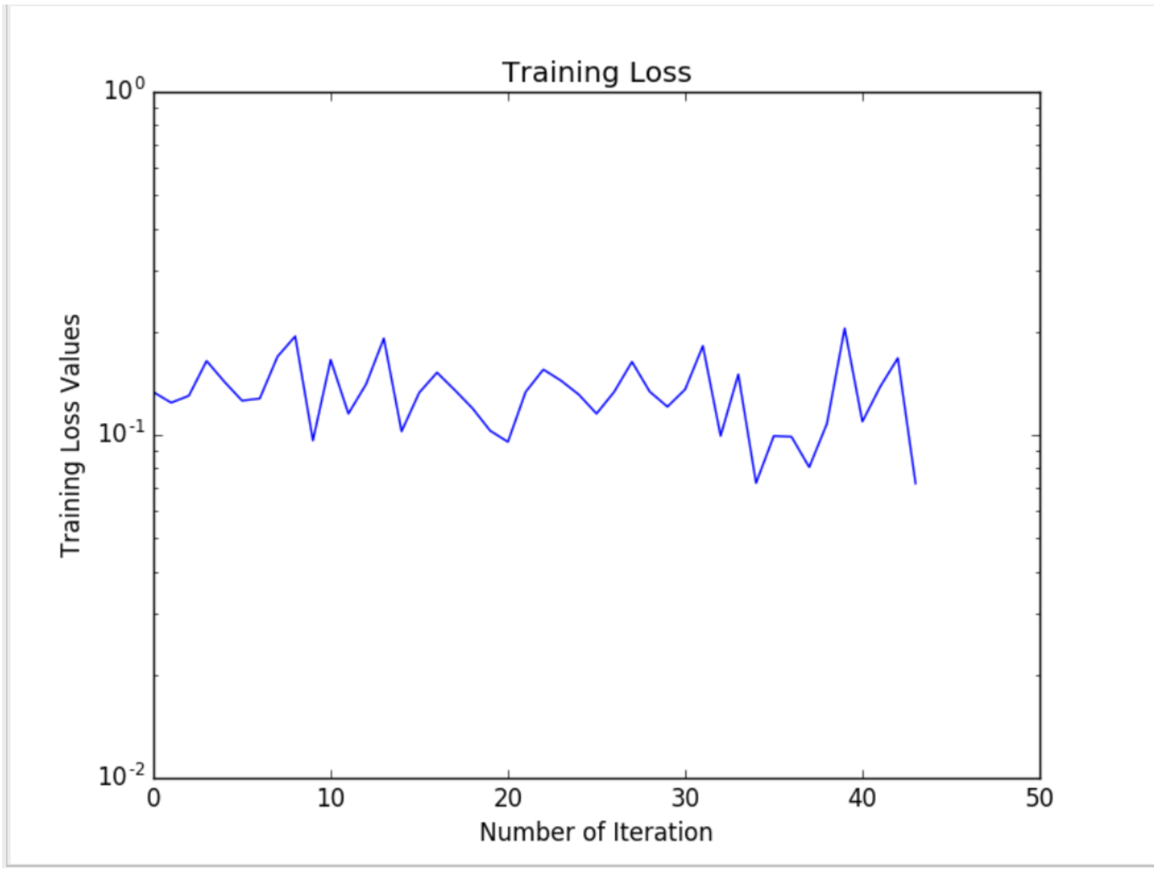
```
classes                                                    ...
airplane    727.0   93.155433   19.365537   54.0  ...
car         968.0  100.000000    0.000000  100.0  ...
cat         885.0  303.732203   95.744543   73.0  ...
dog         702.0  311.933048   99.654237   50.0  ...
flower      843.0  328.167260  189.871326   59.0  ...
fruit      1000.0  100.000000    0.000000  100.0  ...
motorbike   788.0  109.114213   12.083767   66.0  ...
person      986.0  255.981744    0.330622  250.0  ...

[8 rows x 16 columns]
5519 690 690
torch.Size([128, 3, 224, 224]) torch.Size([128])
Test Accuracy of the model on the 690 test images: 90 %
Predicted  0  1  2  3   4   5  6  7  All
Actual
0          4  0  0  0   0   0  0  0    4
1          0  6  0  1   0   0  0  0    7
2          0  0  2  1   0   0  0  0    3
3          0  0  1  4   0   0  0  0    5
4          0  0  0  0  11   0  0  0   11
5          0  0  0  0   0  10  0  0   10
6          0  1  0  0   0   0  2  0    3
7          0  0  0  0   0   0  0  7    7
All        4  7  3  6  11  10  2  7   50
```

The confusion matrix for the nat_img2.py shows that

1 Misclassification for airplane as flower, 1 mis class of cat as a dog, Dog miscalled twice 1 as cat and 1 as car.

Training Loss for 2-Layer CNN:



After that, we train a 3-layer CNN using the same parameters with the architecture shown in Figure 3. The training accuracy is 98.8%, which is close to the 2-layer network. However, the validation accuracy is 88.4%, which is worse than the performance of the 2-layer CNN.

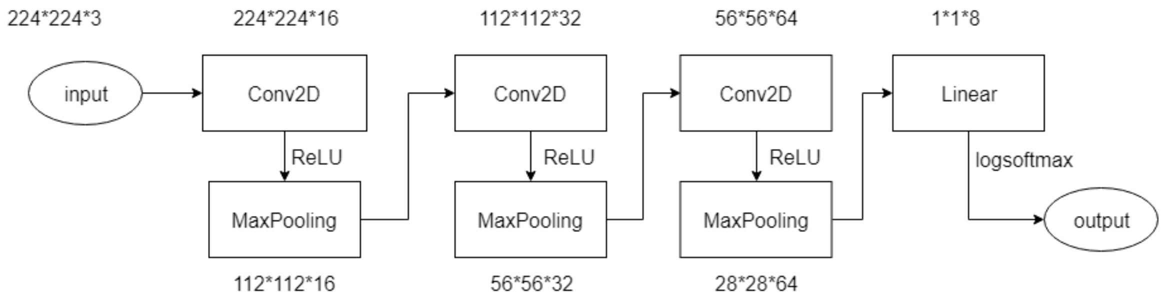


Figure 3. Architecture of a 3-layer CNN

Then, adding another layer gives a 4-layer CNN. We also experiment on a variant of that using only 2 max pooling layers instead of 4. The result of the models is shown in Table 2. For the first trial using 5 epochs, the accuracy on the training set is low, which means the model is underfitting the data. Therefore, we increase the number of epochs to 8. As a result, the training accuracy increased but the validation accuracy decreased. The variant of the 4-layer CNN performs worse on the validation dataset than other models.

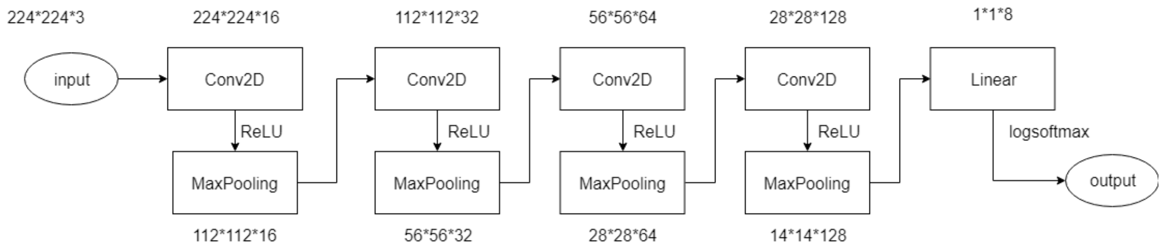


Figure 4-1. Architecture of a 4-layer CNN

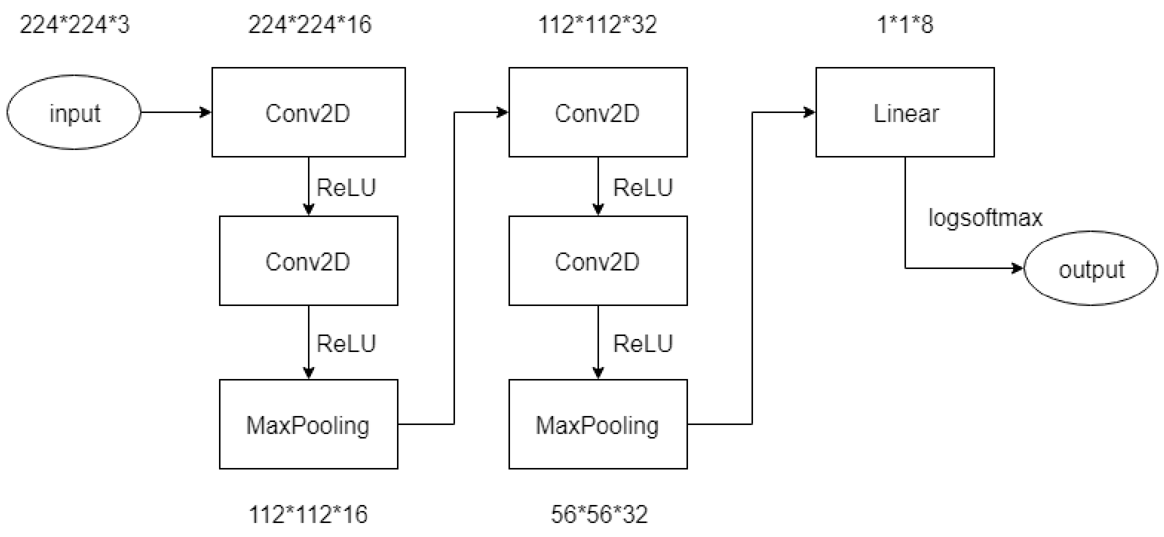


Figure 4-2. Architecture of a 4-layer CNN (Variant)

Table 2. Accuracy of a 4-layer CNN on training and validation dataset

Number of Max Pooling Layers	**Number of epochs**	**Training Accuracy (%)**	**Validation Accuracy (%)**
4	5	92.8	88.7
4	8	96.1	87.7
2	5	98.5	86.2

The next model is a 5-layer CNN shown in Figure 5 and the results in Table 3. Without using dropout layer, the model gives the best performance so far with a validation accuracy of 91.4% using 8 epochs. Since it is the model gives the highest validation accuracy so far. We tested on using an additional dropout layer on the model. However, the two models using a dropout layer does not perform so well as without using it.

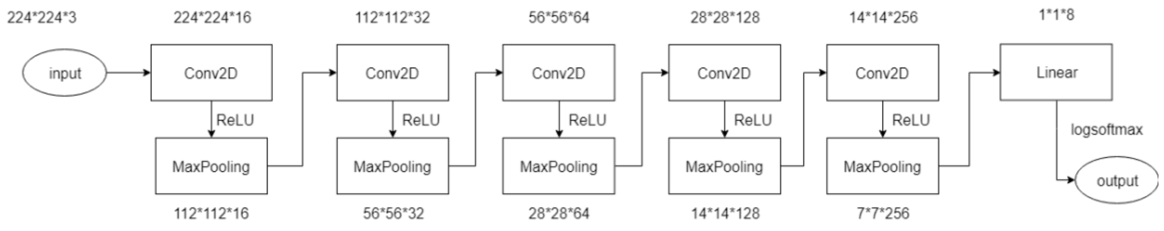


Figure 5. Architecture of a 5-layer CNN

Table 3. Accuracy of a 5-layer CNN on training and validation dataset

Number of epochs	**Dropout Rate**	**Training Accuracy (%)**	**Validation Accuracy (%)**
5	0	92.0	89.1
8	0	97.6	91.4
10	0	95.3	86.7
8	0.2	99.5	89.9
8	0.4	95.6	90.9

Lastly, we tried to experiment on a 6-layer CNN. Adding another convolution and max pooling after 5-layer CNN does not make much sense, because the dimension of the output would be too small. Using a structure similar to Figure 4-2 with 6 convolution layers and 3 max pooling layers will run out of GPU memory.

After choosing the best model, we use test dataset to provide an unbiased evaluation on the 5-layer model. The test accuracy is 91.2%, which is close to the validation accuracy as expected.

Summary and Conclusions

In conclusion, the best model we have is a 5-layer CNN with learning rate = 1e-03, number of epochs = 8, and dropout rate = 0. The test accuracy of this model is 91.2%. Accuracies of all CNN models trained are shown in Table 4.

Table 4. Accuracies of all CNN models on training and validation dataset

Number of Convolution Layer	Number of Max Pooling Layer	Learning Rate	Dropout Rate	Number of epochs	Training Accuracy	Validation Accuracy
2	2	1e-02	0	5	91.4	83.8
2	2	1e-03	0	5	98.6	90.2
2	2	1e-04	0	5	98.4	66.7
3	3	1e-03	0	5	98.2	88.4
4	4	1e-03	0	5	92.8	88.7
4	4	1e-03	0	8	96.1	87.7
4	2	1e-03	0	5	98.5	86.2
5	5	1e-03	0	5	92.0	89.1
5	5	1e-03	0	8	97.6	91.4
5	5	1e-03	0	10	95.3	86.7
5	5	1e-03	0.2	8	99.5	89.9
5	5	1e-03	0.4	8	95.6	90.9

What we have learned is all parameters have some effect on the model, and there are interactions among most of the parameters. With the same parameters, a CNN model has more layers does not necessarily perform better. Also, using more number epochs does not always increase the accuracy on the training dataset. In our experiments, using a dropout layer improves the training accuracy, but not the validation accuracy. Changing other parameters along with the dropout layer may give a better model.

One of the improvements could be make is do to more experiments on the parameters that were not tested. Also, the combinations of the parameters that were not tested may provide more information on the effects. Lastly, if we had more GPU memory, we could test the model with more convolution layers and different architecture of networks.

Reference

[1] Roy, Prasun. “Natural Images.” *Kaggle*, 11 Aug. 2018, www.kaggle.com/prasunroy/natural-images.

[2] “NN-SVG.” *NN SVG*, alexlenail.me/NN-SVG/LeNet.html.

[3] “Pytorch VGG16 Natural Images.” *Kaggle*, www.kaggle.com/gabrielloye/pytorch-vgg16-natural-images.

[4] “VGG16 - Convolutional Network for Classification and Detection.” *VGG16 - Convolutional Network for Classification and Detection*, 21 Nov. 2018, neurohive.io/en/popular-networks/vgg16/.

Appendix

Sample Code of a 5-layer CNN

```
from torchvision import transforms, datasets, models
import torch
from torch import optim, cuda
from torch.utils.data import DataLoader, sampler, random_split
import torch.nn as nn
from PIL import Image
import numpy as np
import pandas as pd
import os
import matplotlib.pyplot as plt
from torch.autograd import Variable

print(os.listdir('./natural_images'))

# Visualising Data
classes = []
img_classes = []
n_image = []
height = []
width = []
dim = []

# Using folder names to identify classes
for folder in os.listdir('./natural_images'):
    classes.append(folder)

    # Number of each image
    images = os.listdir('./natural_images/' + folder)
    n_image.append(len(images))

    for i in images:
        img_classes.append(folder)
        img = np.array(Image.open('./natural_images/' + folder + '/' + i))
        height.append(img.shape[0])
        width.append(img.shape[1])
    dim.append(img.shape[2])

df = pd.DataFrame({
    'classes': classes,
    'number': n_image,
    "dim": dim
})
print("Random heights:" + str(height[10]), str(height[123]))
print("Random Widths:" + str(width[10]), str(width[123]))
df

image_df = pd.DataFrame({
    "classes": img_classes,
    "height": height,
    "width": width
})
img_df = image_df.groupby("classes").describe()
print(img_df)

image_transforms = {
    # Train uses data augmentation
    'train':
    transforms.Compose([
        transforms.RandomResizedCrop(size=256, scale=(0.95, 1.0)),
        transforms.RandomRotation(degrees=15),
        transforms.ColorJitter(),
        transforms.RandomHorizontalFlip(),
        transforms.CenterCrop(size=224),  # Image net standards
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])  # Imagenet standards
    ]),
    'val':
    transforms.Compose([
        transforms.Resize(size=256),
        transforms.CenterCrop(size=224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'test':
    transforms.Compose([
        transforms.Resize(size=256),
        transforms.CenterCrop(size=224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

def imshow_tensor(image, ax=None, title=None):

    if ax is None:
        fig, ax = plt.subplots()

    # Set the color channel as the third dimension
    image = image.numpy().transpose((1, 2, 0))

    # Reverse the preprocessing steps
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    image = std * image + mean

    # Clip the image pixel values
    image = np.clip(image, 0, 1)

    ax.imshow(image)
    plt.axis('off')

    return ax, image
#--------------------------------
# Hyperparameters Defined
```

```python
#-------------------------------
batch_size = 128

#input_size = 1000
#hidden_size = 120
#num_classes = 8
num_epochs = 8
#batch_size = 64
learning_rate =1e-3
#----------------------------------------------------------------------------
all_data = datasets.ImageFolder(root='./natural_images')
train_data_len = int(len(all_data)*0.8)
valid_data_len = int((len(all_data) - train_data_len)/2)
test_data_len = int(len(all_data) - train_data_len - valid_data_len)
train_data, val_data, test_data = random_split(all_data, [train_data_len, valid_data_len,
test_data_len])
train_data.dataset.transform = image_transforms['train']
val_data.dataset.transform = image_transforms['val']
test_data.dataset.transform = image_transforms['test']
print(len(train_data), len(val_data), len(test_data))

train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_data, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=True)

trainiter = iter(train_loader)
images, labels = next(trainiter)
print(images.shape, labels.shape)

#----------------------------------------------------------------------------------------------
# Class for 2 convolutinal - layer network
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=(3, 3), padding=(1, 1)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False))
        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=(3, 3), padding=(1, 1)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False))
        self.layer3 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=(3, 3), padding=(1, 1)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False))
        self.layer4 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=(3, 3), padding=(1, 1)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False))
        self.layer5 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=(3, 3), padding=(1, 1)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False))
        self.fc = nn.Sequential(
            nn.Linear(7 * 7 * 256, 8),
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.layer5(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        out = self.softmax(out)
        return out

#-----------------------------------------------------------------------------------------------
--
cnn = CNN()
cnn.cuda()

# Loss and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn.parameters(), lr=learning_rate)

# Train the Model
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = Variable(images).cuda()
        labels = Variable(labels).cuda()

        # Forward + Backward + Optimize
        optimizer.zero_grad()
        outputs = cnn(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        print('Epoch [%d/%d], Iter [%d/%d] Loss: %.4f'
              % (epoch + 1, num_epochs, i + 1, (len(train_data) // batch_size) + 1,
loss.item()))
# ----------------------------------------------------------------------------------
# Test the Model
cnn.eval()  # Change model to 'eval' mode (BN uses moving mean/var).
correct = 0
total = 0
for images, labels in train_loader:
    images = Variable(images).cuda()
    labels = Variable(labels).cuda()
    outputs = cnn(images)
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum()
# ----------------------------------------------------------------------------------
print('Test Accuracy of the model on Training Data: %d %%' % (100 * correct / total))
# ----------------------------------------------------------------------------------
correct = 0
```

```
total = 0
for images, labels in val_loader:
    images = Variable(images).cuda()
    labels = Variable(labels).cuda()
    outputs = cnn(images)
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum()
# ---------------------------------------------------------------------------------
print('Test Accuracy of the model on Validation Data: %d %%' % (100 * correct / total))
# ---------------------------------------------------------------------------------

correct = 0
total = 0
for images, labels in test_loader:
    images = Variable(images).cuda()
    labels = Variable(labels).cuda()
    outputs = cnn(images)
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum()
# ---------------------------------------------------------------------------------
print('Test Accuracy of the model on Test Data: %d %%' % (100 * correct / total))
# ---------------------------------------------------------------------------------
torch.cuda.empty_cache()
```