DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING, UCLA
ECE 211A: DIGITAL IMAGE PROCESSING I

INSTRUCTOR: Prof. Achuta Kadambi
TA: Yunhao Ba

NAME: Qiong Hu
UID: 405065032

HOMEWORK 3: CONVOLUTIONAL NEURAL NETWORKS

QUESTION	TOPIC	MAX. POINTS	GRADED POINTS	REMARKS
1	Dropout	2.0		
2	Weight Decay and L2 Regularization	2.0		
3	Data Augmentation	2.0		
4	Overfitting CIFAR-10	2.0		
5	Preventing Overfitting	2.0		
	Total	10.0		

1 Dropout

Dropout is one of the common techniques to prevent overfitting in deep learning. Read the dropout paper [2], briefly describe how dropout layer works at the training time and the testing time, and explain the intuition behind these arrangements.

At the training time, a unit at the dropout layer is present with probability p and is connected to units in the next layer with weights $\mathbf{w}$. At the testing time, the unit is always present and the weights are multiplied by p, becoming $p\mathbf{w}$.

The intuition behind dropout comes from a theory of the role of sex in evolution, where sexual reproduction involves random half of the genes from each parents and asexual reproduction is a slightly mutated copy of the parent. The sexual reproduction has its superiority in that the ability of a set of genes to be able to work well with another random set of genes makes them more robust and less dependent on each other. Similarly, each hidden unit in a neural network trained with dropout must learn to work with a randomly chosen sample of other units, making each hidden unit more robust and driving it towards creating useful features on its own without relying on other hidden units to correct its mistakes.

2 Weight Decay and L2 Regularization

What is the L2 regularization? What is the weight decay? When will these two methods become equivalent? When will these two methods perform differently? [1] might be helpful for this question.

"Weight decay" is a method that when training neural networks, after each epoch of update, the weights are multiplied by a factor slightly less than one so that the weights will not grow too large. This is a regularization method to reduce model overfitting.

"L2 regularization" is a method that adding the sum of the squared weights in the loss function as a penalty term to be minimized. This is a common method of regularization to reduce overfitting.

L2 regularization and weight decay are equivalent for standard Stochastic Gradient Descent (SGD) with a fixed learning rate, but they are not identical for Adaptive Gradient Algorithms (Adam).

3 Data Augmentation

Explain why we need to perform data augmentation, and name five data augmentation methods that you can use when training a convolutional neural network (CNN) on images.

Data augmentation is able to artificially create variations in existing data to expand the existing data set for CNN to train on. The expanded data set generalizes the features of the signal and potentially reduces the noise in the model, and thereby reduces the overfitting.

Five image augmentation methods of CNN:

1. Flipping: flips the image vertically or horizontally.
2. Rotation: rotates the image by some degree.
3. Cropping: objects appear in different positions in different proportions in the image.
4. Zoom in, zoom out.
5. Changing brightness of contrast.

4 Overfitting CIFAR-10

In this question, you need to build a CNN to conduct a multi-class image classification task on the CIFAR-10 dataset[1] using PyTorch[2]. There are 60000 labeled images of size 32×32 for 10 different classes in this dataset. The training set contains 50000 images, and the testing set contains 10000 images. You should train your model on the training set and perform testing on the testing set. Validation is not required for this homework.

Here are some requirements for this question:

1. You should use the rectified linear unit (ReLU) as your activation function.
2. You should use the cross entropy loss as your loss function.
3. Your CNN should only consist of convolutional layers and fully-connected layers.
4. Your CNN should only contain 6 layers.
5. The model accuracy on the training set should be around 100%.

Here are some useful resources for your implementation:

1. PyTorch tutorial on CIFAR-10 classification: `https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html`
2. Google Colab for computational resource: `https://colab.research.google.com/`

Report the architecture of your CNN, and plot the training loss history across all the training epochs. Provide your model accuracy on both the training set and the testing set. You should also provide the accuracy for each individual class on these two sets. Place the answers for the above questions in the box below, and place your code in Appendix A.

The architecture is as follows:

Convolutional layer (in_channels = 3, out_channels = 8, kernel_size = 3)
$\Rightarrow$ ReLU $\Rightarrow$ Maxpool
$\Rightarrow$ Convolutional layer (in_channels = 8, out_channels = 32, kernel_size = 2)
$\Rightarrow$ ReLU $\Rightarrow$ Maxpool
$\Rightarrow$ Convolutional layer (in_channels = 32, out_channels = 64, kernel_size = 2)
$\Rightarrow$ ReLU $\Rightarrow$ Maxpool
$\Rightarrow$ Fully-connected layer (in_channels = 64*3*3, out_channels = 256)
$\Rightarrow$ ReLU
$\Rightarrow$ Fully-connected layer (in_channels = 256, out_channels = 84)
$\Rightarrow$ ReLU
$\Rightarrow$ Fully-connected layer (in_channels = 84, out_channels = 10)

[1] `https://www.cs.toronto.edu/~kriz/cifar.html`
[2] `https://pytorch.org/`

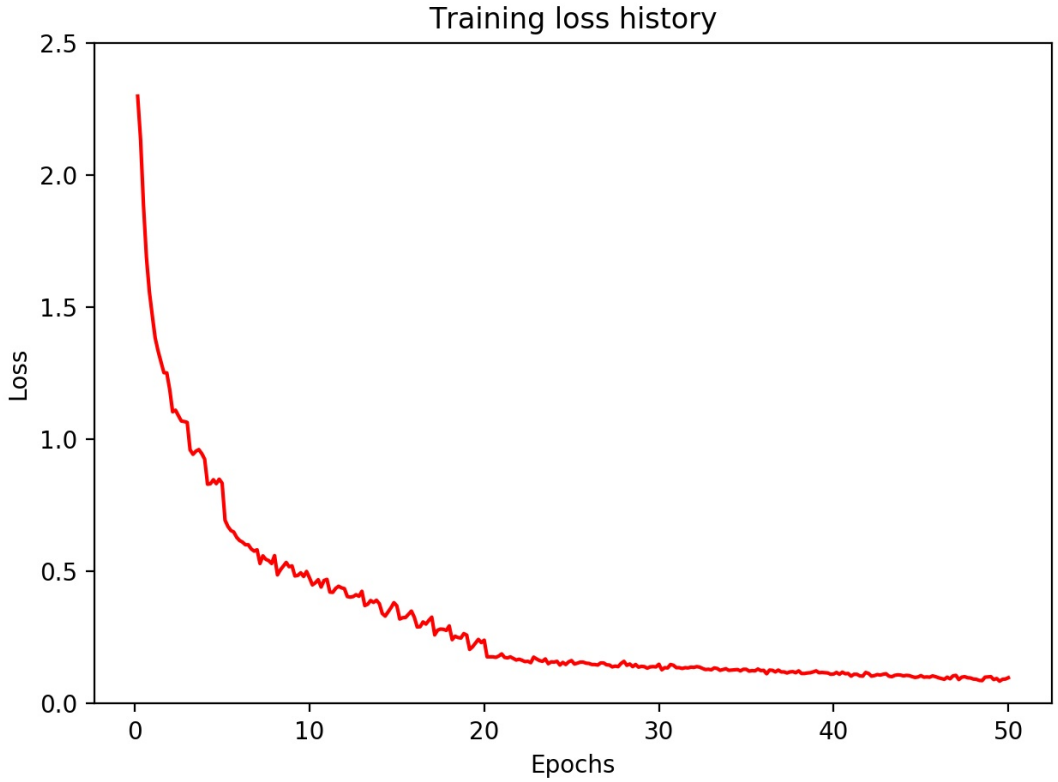

Figure 1: Training loss history across all 50 training epochs.

Average model accuracy on the training set is: 98.38%
Average model accuracy on the testing set is: 72.04%

Model accuracy on each individual class in testing set is:

- Accuracy of plane: 75 %
- Accuracy of car: 83 %
- Accuracy of bird: 63 %
- Accuracy of cat: 51 %
- Accuracy of deer: 67 %
- Accuracy of dog: 63 %
- Accuracy of frog: 79 %
- Accuracy of horse: 74 %
- Accuracy of ship: 82 %
- Accuracy of truck: 79 %

5 Preventing Overfitting

Try various methods to prevent overfitting, and improve your accuracy on the testing set to be around 80%. You should use the same architecture as in Question 4, however, additional dropout layers are allowed. Report your model accuracy on both the training set and the testing set with per-class details as in Question 4, and plot the training loss history across all the training epochs. Discuss the methods you used to achieve this testing accuracy, and summarize your findings briefly. Place the answers for the above questions in the box below, and place your code in Appendix B.

The architecture is as follows (added three dropouts):

Convolutional layer (in_channels = 3, out_channels = 8, kernel_size = 3)
$\Rightarrow$ ReLU $\Rightarrow$ Maxpool
$\Rightarrow$ Convolutional layer (in_channels = 8, out_channels = 32, kernel_size = 2)
$\Rightarrow$ ReLU $\Rightarrow$ Maxpool
$\Rightarrow$ Convolutional layer (in_channels = 32, out_channels = 64, kernel_size = 2)
$\Rightarrow$ ReLU $\Rightarrow$ Maxpool
$\Rightarrow$ Dropout $\Rightarrow$ Fully-connected layer (in_channels = 64*3*3, out_channels = 256)
$\Rightarrow$ ReLU
$\Rightarrow$ Dropout $\Rightarrow$ Fully-connected layer (in_channels = 256, out_channels = 84)
$\Rightarrow$ ReLU
$\Rightarrow$ Dropout $\Rightarrow$ Fully-connected layer (in_channels = 84, out_channels = 10)

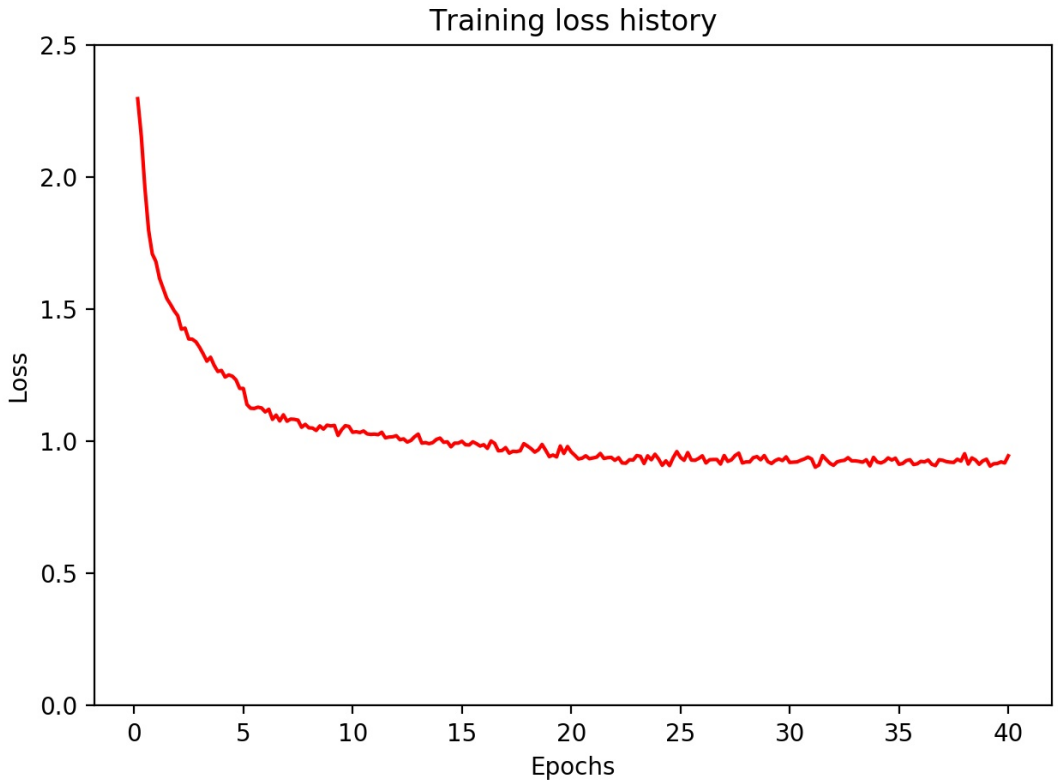

Figure 2: Training loss history across all 50 training epochs.

Average model accuracy on the training set is: 67.63%
Average model accuracy on the testing set is: 64.50%

Model accuracy on each individual class in testing set is:

- Accuracy of plane: 68 %
- Accuracy of car: 77 %
- Accuracy of bird: 52 %
- Accuracy of cat: 46 %
- Accuracy of deer: 57 %
- Accuracy of dog: 52 %
- Accuracy of frog: 74 %
- Accuracy of horse: 69 %
- Accuracy of ship: 75 %
- Accuracy of truck: 73 %

I have tried many different architectures with different number of layers and number of in_channels or out_channels in each layers, whether or not add Dropout layer, where to add dropout layers (whether or not before convolutional layers, and fully-connected layers, before or after the ReLU steps). However, no matter what, the results after adding dropout is always worse than the results as shown in Question 4. It might be the parameters are not correct or the training

epochs are not large enough (due to the slow training speed), or it might also be that the model from Question 4 is actually not overfitting, therefore adding dropout layer reduce the signal information, and thus making the model performance worse.

A Code for Question 4

```python
import torch
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5
  , 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='.', train = True, download = True,
   transform = transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size = 4, shuffle =
  True, num_workers = 8)
testset = torchvision.datasets.CIFAR10(root='.', train = False, download = True,
   transform = transform)
testloader = torch.utils.data.DataLoader(testset, batch_size = 4, shuffle =
  False, num_workers = 8)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship'
  , 'truck')

import matplotlib.pyplot as plt
import numpy as np

def imshow(img):
  img = img/2+0.5   # Denormalize
  npimg = img.numpy()
  plt.imshow(np.transpose(npimg, (1, 2, 0)))
  plt.show()
```

```python
# Define a CNN
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.pool = nn.MaxPool2d(2, 2)
    self.conv1 = nn.Conv2d(3, 8, 3)
    self.conv2 = nn.Conv2d(8, 32, 2)
    self.conv3 = nn.Conv2d(32, 64, 2)
    self.fc1 = nn.Linear(64*3*3, 256)
    self.fc2 = nn.Linear(256, 84)
    self.fc3 = nn.Linear(84, 10)
    self.drop = nn.Dropout2d(p=0.5)

  def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = self.pool(F.relu(self.conv3(x)))
    x = x.view(-1, 64*3*3)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x

net = Net()
```

```
import torch.optim as optim
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr = 0.001, momentum = 0.9)

# Train the network
for epoch in range(50):
  running_loss = 0.0
  for i, data in enumerate(trainloader, 0):
    # Get the inputs; data is a list of [inputs, labels]
    inputs, labels = data

    # Zero the parameter gradients
    optimizer.zero_grad()

    # Forward + backward + optimize
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    if epoch==5:
      optimizer = optim.SGD(net.parameters(), lr = 0.0001, momentum = 0.9)
    elif epoch==20:
      optimizer = optim.SGD(net.parameters(), lr = 0.00001, momentum = 0.9)
    optimizer.step()

    # Print statistics
    running_loss += loss.item()
    if i%2000 == 1999:    # Print every 2000 mini-patches
      print('[%d, %5d] loss: %0.3f' %(epoch+1, i+1, running_loss/2000))
      running_loss = 0.0
```

```
# To see the performance of the network on the whole training dataset
correct = 0
total = 0
with torch.no_grad():
  for data in trainloader:
    images, labels = data
    outputs = net(images)
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum().item()

print('Accuracy of the network on the %d training images is: %0.2f %%' % (total,
   100*correct/total))

# To see the performance of the network on the whole test dataset
correct = 0
total = 0
with torch.no_grad():
  for data in testloader:
    images, labels = data
    outputs = net(images)
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum().item()

print('Accuracy of the network on the %d test images is: %0.2f %%' % (total, 100
  *correct/total))
```

```
# Performace of the network on each class
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
  for data in testloader:
    images, labels = data
    outputs = net(images)
    _, predicted = torch.max(outputs, 1)
    c = (predicted == labels).squeeze()
    for i in range(4):
      label = labels[i]
      class_correct[label] += c[i].item()
      class_total[label] += 1

for i in range(10):
  print('Accuracy of %5s: %2d %%' % (classes[i], 100*class_correct[i]/
    class_total[i]))
```

B Code for Question 5

Everything remains the same as the previous situation except for the following part (Adds dropout layers):

```
class Net(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.pool = nn.MaxPool2d(2, 2)
    self.conv1 = nn.Conv2d(3, 8, 3)
    self.conv2 = nn.Conv2d(8, 32, 2)
    self.conv3 = nn.Conv2d(32, 64, 2)
    self.fc1 = nn.Linear(64*3*3, 256)
    self.fc2 = nn.Linear(256, 84)
    self.fc3 = nn.Linear(84, 10)
    self.drop = nn.Dropout2d(p=0.5)

  def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = self.pool(F.relu(self.conv3(x)))
    x = x.view(-1, 64*3*3)
    x = self.drop(x)
    x = F.relu(self.fc1(x))
    x = self.drop(x)
    x = F.relu(self.fc2(x))
    x = self.drop(x)
    x = self.fc3(x)
    return x

net = Net()
```

References

[1] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[2] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.