Name - Rahul Keshwani NetID - ryk248

Name - Arun Kodnani NetId - ak6384

In [1]:
```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from torch.autograd import Variable
from torch.utils.data.sampler import SubsetRandomSampler
```

In [2]:
```
transform = transforms.Compose([transforms.ToTensor(),transforms.Normalize((

#Getting the data and applying transformation
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=

indices = list(range(len(trainset)))
split = (int)(0.1*len(trainset))

#Splitting indices for train and validation
validation_indices = np.random.choice(indices, size=split, replace=False)
train_indices = list(set(indices) - set(validation_indices))

train_sampler = SubsetRandomSampler(train_indices)
validation_sampler = SubsetRandomSampler(validation_indices)

trainloader = torch.utils.data.DataLoader(trainset, batch_size=100, num_work
validationloader = torch.utils.data.DataLoader(trainset, batch_size=1, num_w
testloader = torch.utils.data.DataLoader(testset, shuffle=True, batch_size=1
```

```
Files already downloaded and verified
Files already downloaded and verified
```

We are here creating a dense CNN instead of a small network with very large hidden layers because that would save the number of parameters of the network.

Following is a paper on ImageNet that we have used as a reference for our architecture. [http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf (http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf)](http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf)

In [3]:
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        #Convolutional layer 1. Input channels=3, Output channels=16.
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)

        #Convolutional layer 2. Input channels=16, Output channels=32.
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)

        #Convolutional layer 3. Input channels=32, Output channels=64
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)

        #Convolutional layer 4. Input channels=64, Output channels=128
        self.conv4 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)

        #Convolutional layer 5. Input channels=128, Output channels=256
        self.conv5 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)

        # Pooling layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

        #Fully connected layer 1
        self.fc1 = nn.Linear(256 * 4 * 4, 1024)

        #Fully connected layer 2
        self.fc2 = nn.Linear(1024, 256)

        #Fully connected layer 3
        self.fc3 = nn.Linear(256, 10)

    def forward(self, x):
        #Convolution 1 and activation. Size changes from (3,32,32) to (16,3
        x = F.relu(self.conv1(x))

        #Downsampling. Size changes from (16,32,32) to (16,16,16)
        x = self.pool(x)

        #Convolution 2 and activation. Size changes from (16,16,16) to (32,
        x = F.relu(self.conv2(x))

        #Downsampling. Size changes from (32,16,16) to (32,8,8)
        x = self.pool(x)

        #Convolution 3 and activation. Size changes from (32,8,8) to (64,8,
        x = F.relu(self.conv3(x))

        #Convolution 4 and activation. Size changes from (64,8,8) to (128,8,
        x = F.relu(self.conv4(x))

        #Convolution 5 and activation. Size changes from (128,8,8) to (256,
        x = F.relu(self.conv5(x))

        #Downsampling. Size changes from (256,8,8) to (256,4,4)
        x = self.pool(x)

        #Flatten data for fully connected layer. size changes from (256,4,4)
```

```
        x = x.view(-1,256 * 4 * 4)

        #First fully connected layer
        x = F.relu(self.fc1(x))

        #Second fully connected layer
        x = F.relu(self.fc2(x))

        #Third fully connected layer.
        x = self.fc3(x)

        return x
```

For calculating the loss we have used "CrossEntropyLoss" which automatically applies Softmax before calculating the loss and hence we have not added a softmax layer in our architecture. This was one of the main reason we have selected this particular loss function.

We have used "Adam Optimizer" because it works really well with Non-Convex functions where it is very easy to land on the local optima. One of the main reason of using this optimizer is that it maintains a learning rate for each of the parameters and keeps it adaptive.

In [4]:

```
def trainCNNModel(num_epochs, learning_rate):
    #Creating an object of CNN class to build the network
    cnn_model = CNN()

    #Defining the loss function and optimizer
    loss_method = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(cnn_model.parameters(), lr=learning_rate)

    #Check if GPU is available
    if torch.cuda.is_available():
        cnn_model = cnn_model.cuda()
        loss_method = loss_method.cuda()

    #Iterate over the input images multiple times
    for epoch in range(num_epochs):
        training_loss = 0.0
        for i, (train_images, train_labels) in enumerate(trainloader):
            train_images, train_labels = Variable(train_images), Variable(tr

            optimizer.zero_grad()

            train_output = cnn_model.forward(train_images)
            loss = loss_method(train_output, train_labels)
            loss.backward()
            optimizer.step()

            training_loss += loss.item()
            if i % 50 == 49:    # print every 2000 mini-batches
                print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, training_l
                training_loss = 0.0

    return cnn_model
```

In [5]:
```
def testCNNModel(cnn_model):
    total = 0
    correct_predictions = 0
    predictions = []
    #Sets the "requires_grad" flag to False to avoid gradient descent during
    with torch.no_grad():
        #Iterate over the test images
        for i, (test_images, test_labels) in enumerate(testloader):
            test_images, test_labels = Variable(test_images), Variable(test_
            test_output = cnn_model(test_images)
            max_value, prediction_class = torch.max(test_output.data, 1)
            predictions.extend(prediction_class)
            total += test_labels.size(0)
            correct_predictions += torch.sum(prediction_class == test_labels

    return total, np.array(correct_predictions), predictions

def calculateAccuracy(total, correct_predictions):
    return 100 * (correct_predictions/total)
```

In [6]:
```
learning_rates = [0.001]
for lr in learning_rates:
    cnn_model = trainCNNModel(5, lr)
    total, correct_predictions, predictions = testCNNModel(cnn_model)
    accuracy = calculateAccuracy(total, correct_predictions)
    print("Accuracy of the network for learning rate = ", lr, "is", accuracy
```

```
[1,    50] loss: 2.170
[1,   100] loss: 1.937
[1,   150] loss: 1.825
[1,   200] loss: 1.696
[1,   250] loss: 1.589
[1,   300] loss: 1.502
[1,   350] loss: 1.422
[1,   400] loss: 1.389
[1,   450] loss: 1.340
[2,    50] loss: 1.293
[2,   100] loss: 1.271
[2,   150] loss: 1.234
[2,   200] loss: 1.205
[2,   250] loss: 1.161
[2,   300] loss: 1.160
[2,   350] loss: 1.136
[2,   400] loss: 1.115
[2,   450] loss: 1.068
[3,    50] loss: 1.025
[3,   100] loss: 0.995
[3,   150] loss: 0.997
[3,   200] loss: 0.999
[3,   250] loss: 0.959
[3,   300] loss: 0.939
[3,   350] loss: 0.928
[3,   400] loss: 0.948
[3,   450] loss: 0.941
[4,    50] loss: 0.849
[4,   100] loss: 0.853
[4,   150] loss: 0.832
[4,   200] loss: 0.820
[4,   250] loss: 0.785
[4,   300] loss: 0.822
[4,   350] loss: 0.819
[4,   400] loss: 0.796
[4,   450] loss: 0.827
[5,    50] loss: 0.719
[5,   100] loss: 0.685
[5,   150] loss: 0.720
[5,   200] loss: 0.704
[5,   250] loss: 0.716
[5,   300] loss: 0.702
[5,   350] loss: 0.698
[5,   400] loss: 0.688
[5,   450] loss: 0.669
Accuracy of the network for learning rate =  0.001 is 71.65 %
```

In [7]:
```
def save_predictions(filename, y):
    np.save(filename, y)
```

In [8]:
```
save_predictions('ans2-ryk248', predictions)
save_predictions('ans2-ak6384', predictions)
```

In [ ]: