Task 1.1

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms


class CNN(nn.Module):

    def __init__(self):
        super(CNN, self).__init__()

        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 6, kernel_size=5, stride=1, padding=0),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))

        self.layer2 = nn.Sequential(
            nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))

        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)

        out = out.view(-1, 16 * 5 * 5)

        out = F.relu(self.fc1(out))
        out = F.relu(self.fc2(out))
        out = self.fc3(out)

        return out
```

Task 1.2

Output Size:
(64 x 64 x 1): Activation size = 64 * 64 * 1 = 4096

Layer_1: 64 - 5 + 1 = 60 x 60 x 6
MaxPool: (60 - 2) / 2 + 1 = 30 x 30 x 6

Layer_2: 30 - 5 + 1 = 26 x 26 x 16
MaxPool: (30 - 2) / 2 + 1 = 15 x 15 x 16

fc1: 120 x 1
fc2: 84 x 1
fc3: 20 x 1

Number of Parameters:
Layer1:
Weights = 5^2 x 1 x 6 = 150
Biases = 6
Parameters = 150 + 6 = 156

Layer2:
Weights = 5^2 x 1 x 16 = 400
Biases = 16
Parameters = 400 + 16 = 416

fc1 connected to a Conv Layer:
Weights: 15^2 x 16 x 120 = 432000
Biases: 120
Parameters = 432120

fc2 connected to the previous fc layer:
Weights: 120 x 84 = 10080
Biases: 84
Parameters: 10164

fc3 connected to the previous fc layer:
Weights: 84 x 10 = 840
Biases: 10
Parameters: 850

Total Parameters: 156 + 416 + 432120 + 10164 + 850 = 443706

Task 1.3

```
def train(self):
    trainloader, _ = self.load_data()
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(cnn.parameters(), lr=0.001, momentum=0.9)

    for epoch in range(10):
        for i, data in enumerate(trainloader, 0):
            inputs, labels = data

            optimizer.zero_grad()
            outputs = cnn(inputs)

            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

def eval(self):
    _, testloader = self.load_data()
    total_correct = 0
    total_images = 0

    with torch.no_grad():
        for data in testloader:
            images, labels = data
            outputs = cnn(images)
            _, predicted = torch.max(outputs.data, 1)

            total_images += labels.size(0)
            total_correct += (predicted == labels).sum().item()

    model_accuracy = total_correct / total_images * 100
    print('Model accuracy on {0} test images: {1:.2f}%'.format(total_images,
model_accuracy))
```

```python
def load_data(self):
    transform = transforms.Compose(
        [
         transforms.ToTensor()
        ])

    trainset = torchvision.datasets.CIFAR10(root='./Data',
                                            train=True,
                                            download=True,
                                            transform=transform)

    trainloader = torch.utils.data.DataLoader(trainset,
                                              batch_size=16,
                                              shuffle=True)

    testset = torchvision.datasets.CIFAR10(root='./Data',
                                           train=False,
                                           download=True,
                                           transform=transform)

    testloader = torch.utils.data.DataLoader(testset,
                                             batch_size=16,
                                             shuffle=False)

    return trainloader, testloader


if __name__ == "__main__":
    cnn = CNN()
    cnn.train()
    cnn.eval()
```

Model accuracy on 10000 test images: 58.14%

Task 1.4

```python
def load_data(self):
    transform = transforms.Compose(
        [
         transforms.RandomHorizontalFlip(p=0.5),
         transforms.RandomCrop(32, padding=4),
         transforms.ToTensor(),
         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
         ])

    trainset = torchvision.datasets.CIFAR10(root='./Data',
                                            train=True,
                                            download=True,
                                            transform=transform)

    trainloader = torch.utils.data.DataLoader(trainset,
                                              batch_size=16,
                                              shuffle=True)

    testset = torchvision.datasets.CIFAR10(root='./Data',
                                           train=False,
                                           download=True,
                                           transform=transform)

    testloader = torch.utils.data.DataLoader(testset,
                                             batch_size=16,
                                             shuffle=False)

    return trainloader, testloader
```

Input normalization:
Model accuracy on 10000 test images: 61.72%
In comparison to the baseline model its about 3% better.

Input normalization and augmentation:
Model accuracy on 10000 test images: 57.08%
In comparison to the baseline model its about 1% worse.

Task 1.5
0.1: 10.00%
0.01: 51.34%
0.001: 57.08%
0.0001: 32.60%

The learning rate tells the optimizer how far to move the weights in the direction opposite of the gradient. If it is low, then training is more reliable, but optimization will take a lot of time because steps towards the minimum of the loss function are tiny, see above: 0.1 -> 10%.
If the learning rate is high, then training may not converge. Weight changes can be so big that the optimizer overshoots the minimum and makes the loss worse, 0.0001: 32.60%. The best learning rate in the four examples is 0.001 -> 57.08%.

Task 1.6

```
def __init__(self):
    super(CNN, self).__init__()

    self.layer1 = nn.Sequential(
        nn.Conv2d(3, 6, kernel_size=5, stride=1, padding=0),
        nn.BatchNorm2d(6),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2))

    self.layer2 = nn.Sequential(
        nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0),
        nn.BatchNorm2d(16),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2))

    self.fc1 = nn.Linear(16 * 5 * 5, 120)
    self.fc2 = nn.Linear(120, 84)
    self.fc3 = nn.Linear(84, 10)

def forward(self, x):
    out = self.layer1(x)
    out = self.layer2(out)

    out = out.view(-1, 16 * 5 * 5)

    out = F.relu(self.fc1(out))
    out = F.relu(self.fc2(out))
    out = self.fc3(out)

    return out
```

In contrast to the baseline model with 58.14% the model with a batch normalization layer performs a accuracy of 56.33%. So it’s a little bit worse, but it should reduce overfitting because it has slight regularization effects.

Task 1.7

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as transforms
import torchvision


class Block(nn.Module):
    '''Depthwise conv + Pointwise conv'''
    def __init__(self, in_planes, out_planes, stride=1):
        super(Block, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, in_planes, kernel_size=3, stride=stride,
padding=1, groups=in_planes, bias=False)
        self.bn1 = nn.BatchNorm2d(in_planes)
        self.conv2 = nn.Conv2d(in_planes, out_planes, kernel_size=1, stride=1,
padding=0, bias=False)
        self.bn2 = nn.BatchNorm2d(out_planes)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = F.relu(self.bn2(self.conv2(out)))
        return out


class MobileNet(nn.Module):
    # (128,2) means conv planes=128, conv stride=2, by default conv stride=1
    cfg = [64, (128,2), 128, (256,2), 256, (512,2), 512, 512, 512, 512, 512,
(1024,2), 1024]

    def __init__(self, num_classes=10):
        super(MobileNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1,
bias=False)
        self.bn1 = nn.BatchNorm2d(32)
        self.layers = self._make_layers(in_planes=32)
        self.linear = nn.Linear(1024, num_classes)

    def _make_layers(self, in_planes):
        layers = []
        for x in self.cfg:
            out_planes = x if isinstance(x, int) else x[0]
            stride = 1 if isinstance(x, int) else x[1]
            layers.append(Block(in_planes, out_planes, stride))
            in_planes = out_planes
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layers(out)
        out = F.avg_pool2d(out, 2)
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out

    def train(self):
        trainloader, _ = self.load_data()
        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.SGD(mobileNet.parameters(), lr=0.001, momentum=0.9)

        for epoch in range(5):
            for i, data in enumerate(trainloader, 0):
                inputs, labels = data

                optimizer.zero_grad()
                outputs = mobileNet(inputs)

                loss = criterion(outputs, labels)
```

```
                loss.backward()
                optimizer.step()

    def eval(self):
        _, testloader = self.load_data()
        total_correct = 0
        total_images = 0

        with torch.no_grad():
            for data in testloader:
                images, labels = data
                outputs = mobileNet(images)
                _, predicted = torch.max(outputs.data, 1)

                total_images += labels.size(0)
                total_correct += (predicted == labels).sum().item()

        model_accuracy = total_correct / total_images * 100
        print('Model accuracy on {0} test images: {1:.2f}%'.format(total_images,
model_accuracy))

    def load_data(self):
        transform = transforms.Compose(
            [
             transforms.RandomHorizontalFlip(p=0.5),
             transforms.RandomCrop(32, padding=4),
             transforms.ToTensor(),
             transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
             ])

        trainset = torchvision.datasets.CIFAR10(root='./Data',
                                                train=True,
                                                download=True,
                                                transform=transform)

        trainloader = torch.utils.data.DataLoader(trainset,
                                                  batch_size=16,
                                                  shuffle=True)

        testset = torchvision.datasets.CIFAR10(root='./Data',
                                               train=False,
                                               download=True,
                                               transform=transform)

        testloader = torch.utils.data.DataLoader(testset,
                                                 batch_size=16,
                                                 shuffle=False)

        return trainloader, testloader


if __name__ == "__main__":
    mobileNet = MobileNet()
    mobileNet.train()
    mobileNet.eval()
```

Not enough computing power to calculate and get some results…