Angel Nieto García 2211798052

Homework #4

1. Source code

a) Dataloader: I went with the build Dataloader from torch.utils.data for simplicity

```
batch_size = 64

trainloader = torch.utils.data.DataLoader(trainset, batch_size,
                                          shuffle=True, num_workers=2)
valloader = torch.utils.data.DataLoader(validationset, batch_size, num_workers=2)
testloader = torch.utils.data.DataLoader(testset, batch_size,
                                          shuffle=False, num_workers=2)
```

b) Model: Uses additional Conv layers to get better classification and at the end we add a fully connected layer with Dropouts to increase variance in the distribution.

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')   # use GPU
if available

class Net(nn.Module):
    def __init__(self):
      super().__init__()
      self.conv_layer = nn.Sequential(

          # Conv Layer block 1
          nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1),
          nn.BatchNorm2d(32),
          nn.ReLU(inplace=True),
          nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
          nn.ReLU(inplace=True),
          nn.MaxPool2d(kernel_size=2, stride=2),

          # Conv Layer block 2
          nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
          nn.BatchNorm2d(128),
          nn.ReLU(inplace=True),
          nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1),
          nn.ReLU(inplace=True),
          nn.MaxPool2d(kernel_size=2, stride=2),
          nn.Dropout2d(p=0.05),

          # Conv Layer block 3
          nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1),
          nn.BatchNorm2d(256),
          nn.ReLU(inplace=True),
          nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1),
```

```
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )


        self.fc_layer = nn.Sequential(
            nn.Dropout(p=0.1),
            nn.Linear(4096, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 512),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.1),
            nn.Linear(512, 10)
        )


    def forward(self, x):
        """Perform forward."""

        # conv layers
        x = self.conv_layer(x)

        # flatten
        x = x.view(x.size(0), -1)

        # fc layer
        x = self.fc_layer(x)

        return x


    def training_step(self, batch):
        images, labels = batch[0].to(device), batch[1].to(device)
        out = self(images)                  # Generate predictions
        loss = F.cross_entropy(out, labels) # Calculate loss
        return loss

    @torch.no_grad()
    def validation_step(self, batch):
        images, labels = batch[0].to(device), batch[1].to(device)
        out = self(images)                    # Generate predictions
        loss = F.cross_entropy(out, labels)   # Calculate loss
        acc = accuracy(out, labels)           # Calculate accuracy
        return {'val_loss': loss.detach(), 'val_acc': acc.detach()}

    @torch.no_grad()
    def validation_epoch_end(self, outputs):
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()   # Combine losses
        batch_accs = [x['val_acc'] for x in outputs]
```

```
        epoch_acc = torch.stack(batch_accs).mean()      # Combine accuracies
        return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}

    def epoch_end(self, train_loss, epoch, result):
        print("Epoch [{}], train_loss: {:.4f}, val_loss: {:.4f}, val_acc:
{:.4f}".format(epoch, train_loss, result['val_loss'], result['val_acc']))

net = Net()
net = net.to(device)
```

c) Loss Function

```
criterion = nn.CrossEntropyLoss()
```

d) Optimizer

```
optimizer = optim.Adam(net.parameters(), lr=0.001)
```

e) Evaluation

```
# Check the ground truth images
dataiter = iter(testloader)
images, labels = dataiter.next()

imshow(torchvision.utils.make_grid(images[:4]))
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))

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[0].to(device), data[1].to(device)
        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]))
```

f) Training Loop

```
@torch.no_grad()
def evaluate(model, val_loader):
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)

def train(num_epochs, train_loader, val_loader, net, criterion, optimizer):
    history = []
    train_it = 0

    for epoch in range(num_epochs):
        running_loss = 0.0
        for i, data in enumerate(train_loader):
            optimizer.zero_grad()
            # forward
            loss = net.training_step(data)

            # backward
            loss.backward()

            # update the weights
            optimizer.step() # 1 step over optimizer

            running_loss += loss.item()

            train_it += 1

        # Validation phase
        running_loss /= len(train_loader)
        result = evaluate(net, val_loader)
        net.epoch_end(running_loss, epoch, result)
        result['train_loss'] = running_loss
        history.append(result)

    return history
```

g) Datasets and Data augmentation

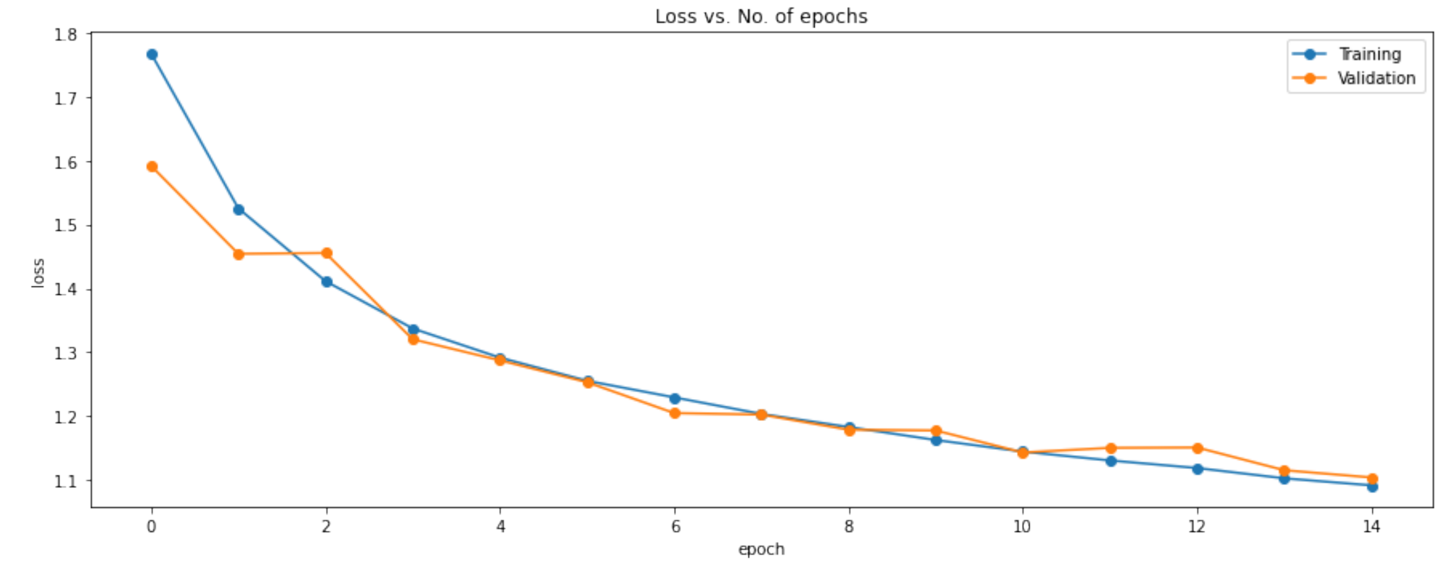
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

train_transform = transforms.Compose([transforms.RandomCrop(32, padding=4,
padding_mode='reflect'),
                         transforms.RandomHorizontalFlip(),
                         transforms.ToTensor(),
                         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5),
inplace=True)])
```

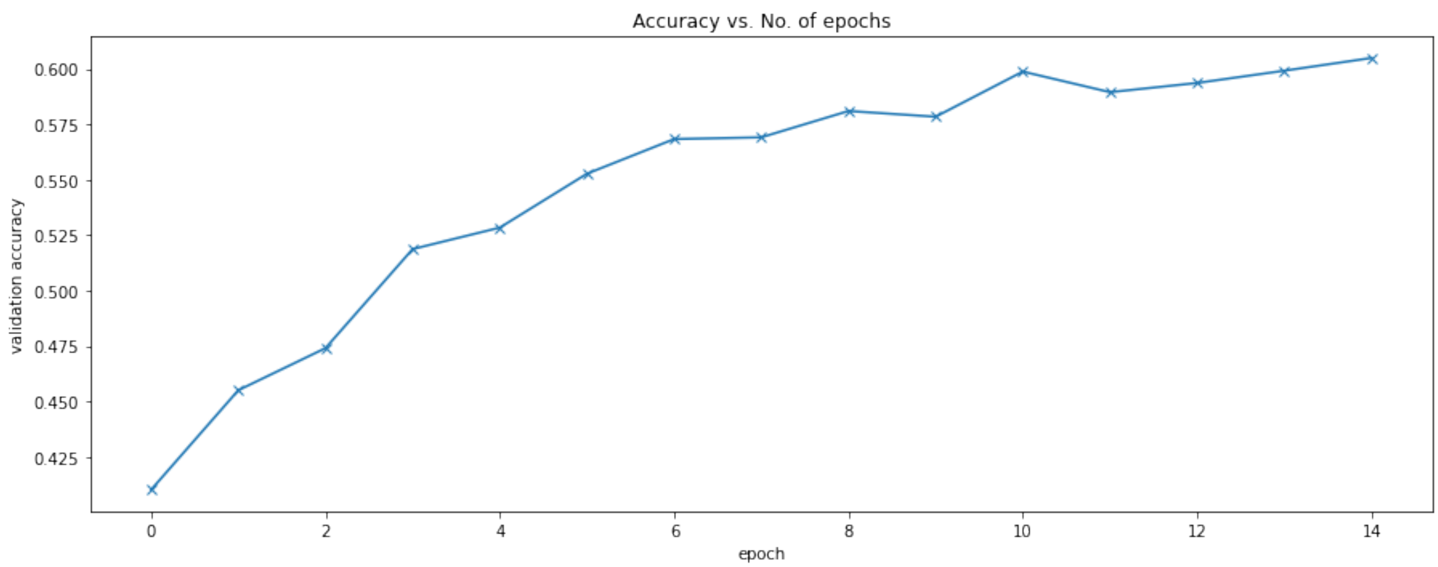
```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=train_transform)


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

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

val_size = 10000
train_size = len(trainset) - val_size

trainset, validationset = random_split(trainset, [train_size, val_size])
len(trainset), len(validationset)
```

2 Discussion

2.1 Evolution of training losses and validation losses with standard LeNet using Adam as optimizer, learning rate = 0.001 and 15 epochs

```
Epoch [0], train_loss: 1.7679, val_loss: 1.5919, val_acc: 0.4102
Epoch [1], train_loss: 1.5254, val_loss: 1.4541, val_acc: 0.4551
Epoch [2], train_loss: 1.4113, val_loss: 1.4554, val_acc: 0.4741
Epoch [3], train_loss: 1.3371, val_loss: 1.3202, val_acc: 0.5188
Epoch [4], train_loss: 1.2914, val_loss: 1.2871, val_acc: 0.5285
Epoch [5], train_loss: 1.2551, val_loss: 1.2529, val_acc: 0.5528
Epoch [6], train_loss: 1.2291, val_loss: 1.2044, val_acc: 0.5684
Epoch [7], train_loss: 1.2031, val_loss: 1.2021, val_acc: 0.5692
Epoch [8], train_loss: 1.1826, val_loss: 1.1784, val_acc: 0.5811
Epoch [9], train_loss: 1.1623, val_loss: 1.1772, val_acc: 0.5785
Epoch [10], train_loss: 1.1441, val_loss: 1.1426, val_acc: 0.5989
Epoch [11], train_loss: 1.1302, val_loss: 1.1499, val_acc: 0.5896
Epoch [12], train_loss: 1.1183, val_loss: 1.1505, val_acc: 0.5938
Epoch [13], train_loss: 1.1022, val_loss: 1.1149, val_acc: 0.5993
Epoch [14], train_loss: 1.0912, val_loss: 1.1035, val_acc: 0.6050
```



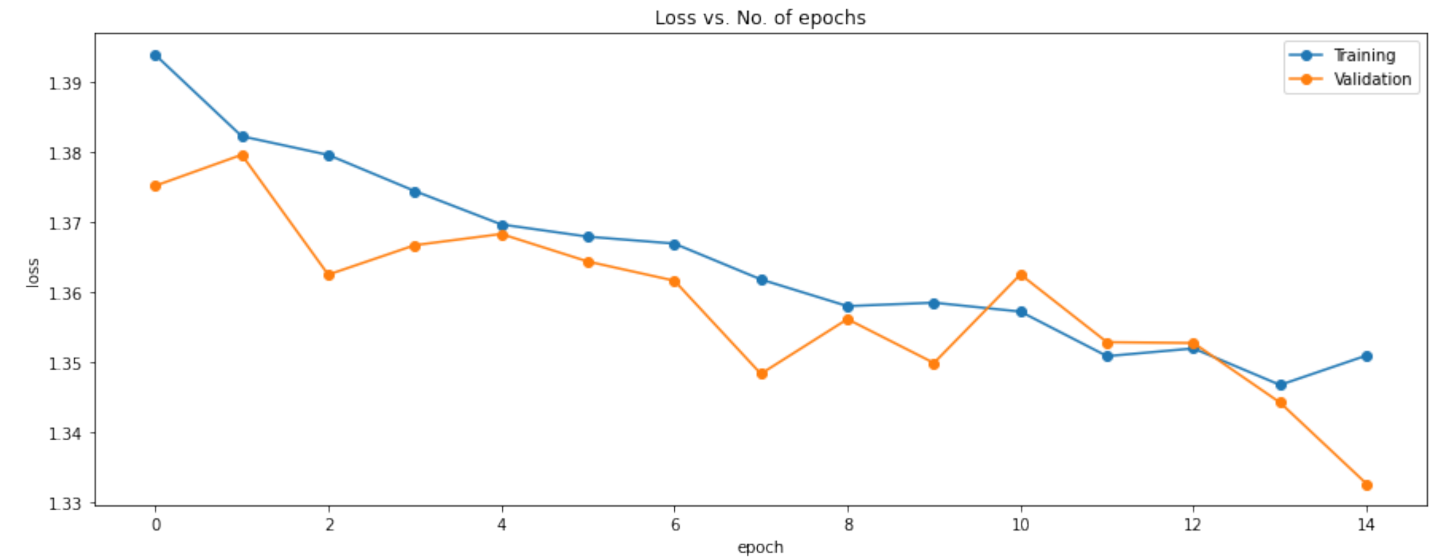


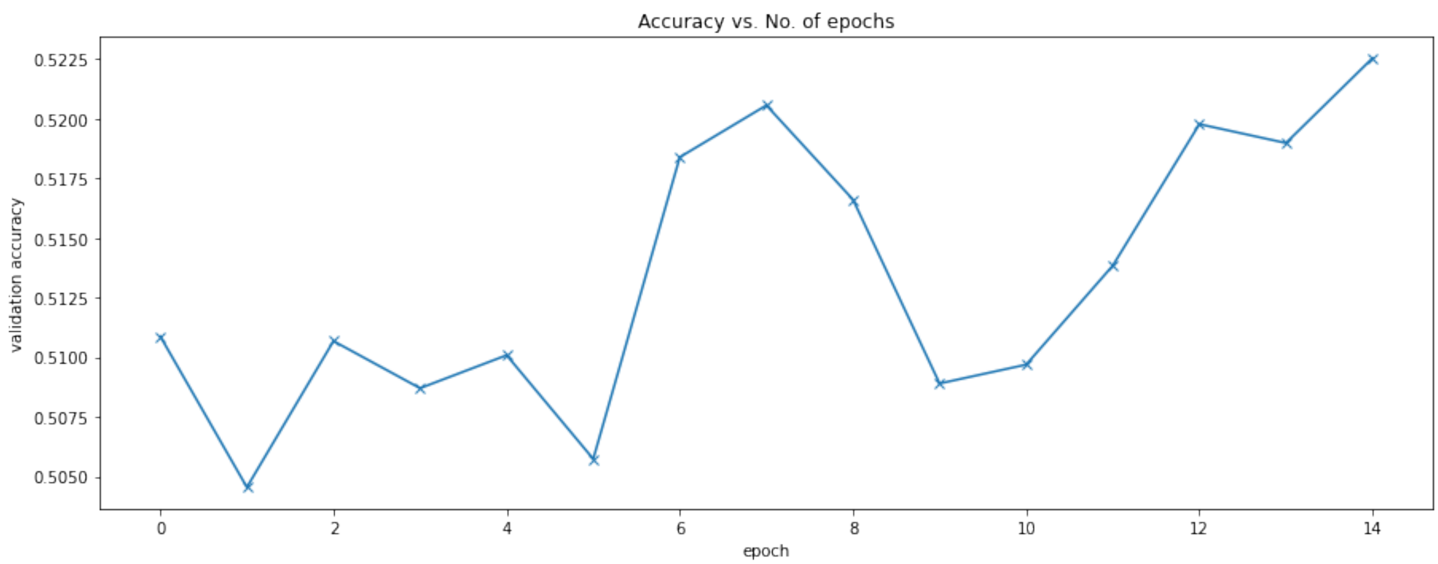
2.2 Effects of parameter choices

2.2.1 SGD Optimizer

As it can be appreciated in the following images a different optimizer such as SGD in augmented data produces worse results since learning is not as effective as with Adam optimizer

```
Epoch [0], train_loss: 1.3938, val_loss: 1.3751, val_acc: 0.5109
Epoch [1], train_loss: 1.3822, val_loss: 1.3796, val_acc: 0.5045
Epoch [2], train_loss: 1.3795, val_loss: 1.3624, val_acc: 0.5107
Epoch [3], train_loss: 1.3744, val_loss: 1.3667, val_acc: 0.5087
Epoch [4], train_loss: 1.3696, val_loss: 1.3683, val_acc: 0.5101
Epoch [5], train_loss: 1.3679, val_loss: 1.3643, val_acc: 0.5057
Epoch [6], train_loss: 1.3669, val_loss: 1.3616, val_acc: 0.5184
Epoch [7], train_loss: 1.3618, val_loss: 1.3483, val_acc: 0.5206
Epoch [8], train_loss: 1.3580, val_loss: 1.3561, val_acc: 0.5166
Epoch [9], train_loss: 1.3584, val_loss: 1.3498, val_acc: 0.5089
Epoch [10], train_loss: 1.3572, val_loss: 1.3625, val_acc: 0.5097
Epoch [11], train_loss: 1.3508, val_loss: 1.3528, val_acc: 0.5138
Epoch [12], train_loss: 1.3519, val_loss: 1.3527, val_acc: 0.5198
Epoch [13], train_loss: 1.3467, val_loss: 1.3442, val_acc: 0.5190
Epoch [14], train_loss: 1.3509, val_loss: 1.3326, val_acc: 0.5225
```



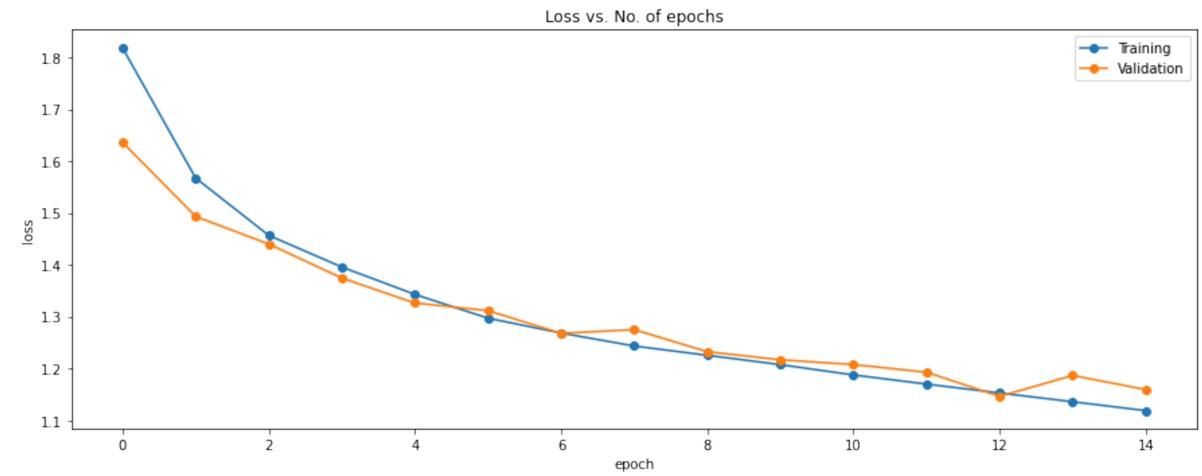


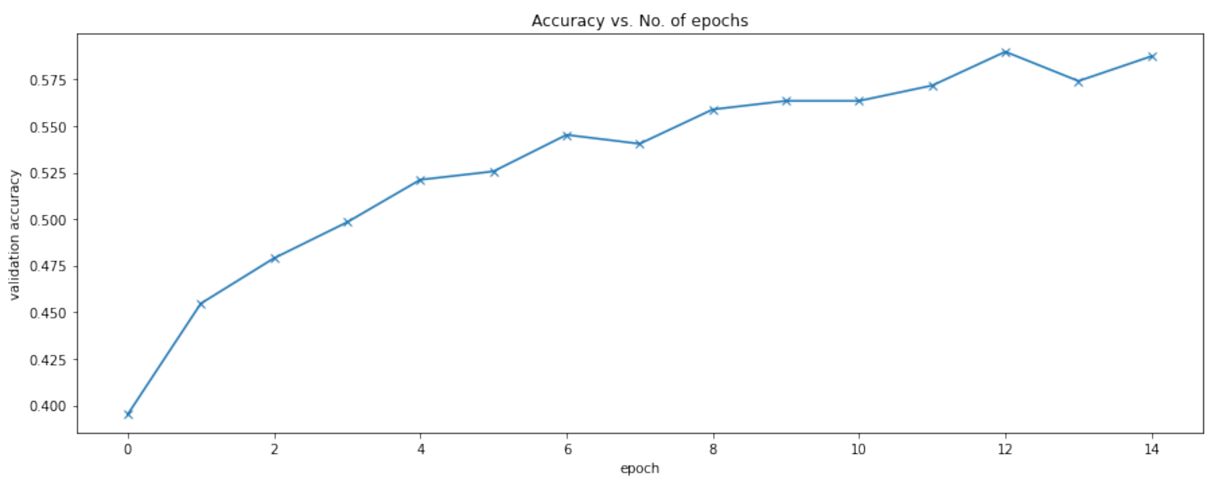
GroundTruth: cat ship ship plane

Overall Accuracy: 55.46%

2.2.2 Using 4x4 filter size instead of 5x5 in Convolutional layers

```
Epoch [0], train_loss: 1.8183, val_loss: 1.6362, val_acc: 0.3952
Epoch [1], train_loss: 1.5674, val_loss: 1.4934, val_acc: 0.4547
Epoch [2], train_loss: 1.4569, val_loss: 1.4402, val_acc: 0.4790
Epoch [3], train_loss: 1.3959, val_loss: 1.3747, val_acc: 0.4983
Epoch [4], train_loss: 1.3431, val_loss: 1.3266, val_acc: 0.5211
Epoch [5], train_loss: 1.2973, val_loss: 1.3119, val_acc: 0.5256
Epoch [6], train_loss: 1.2687, val_loss: 1.2681, val_acc: 0.5452
Epoch [7], train_loss: 1.2437, val_loss: 1.2754, val_acc: 0.5404
Epoch [8], train_loss: 1.2258, val_loss: 1.2327, val_acc: 0.5588
Epoch [9], train_loss: 1.2077, val_loss: 1.2171, val_acc: 0.5634
Epoch [10], train_loss: 1.1879, val_loss: 1.2080, val_acc: 0.5634
Epoch [11], train_loss: 1.1703, val_loss: 1.1932, val_acc: 0.5717
Epoch [12], train_loss: 1.1531, val_loss: 1.1466, val_acc: 0.5898
Epoch [13], train_loss: 1.1362, val_loss: 1.1870, val_acc: 0.5740
Epoch [14], train_loss: 1.1191, val_loss: 1.1597, val_acc: 0.5874
```



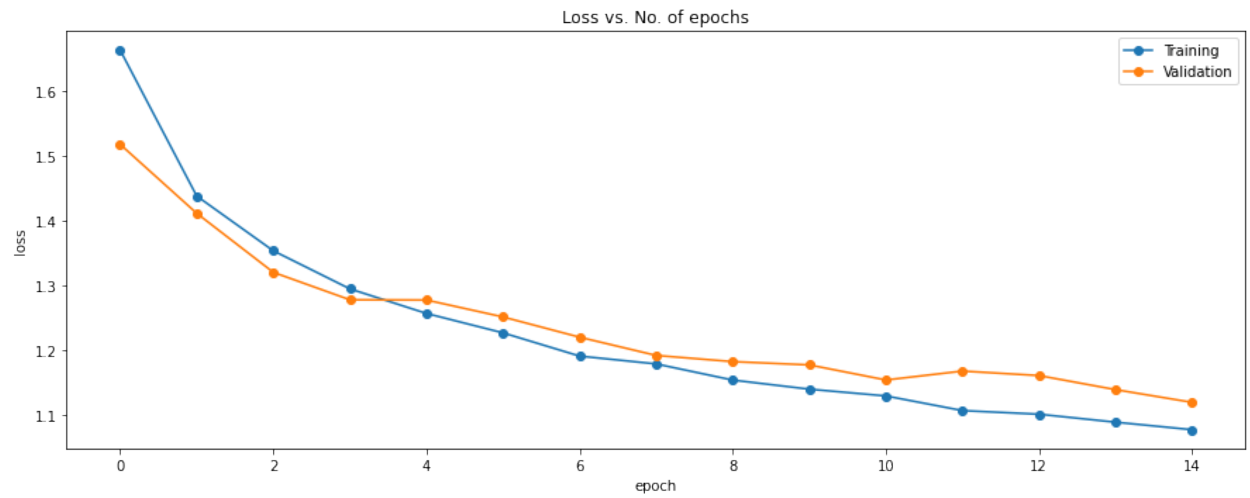


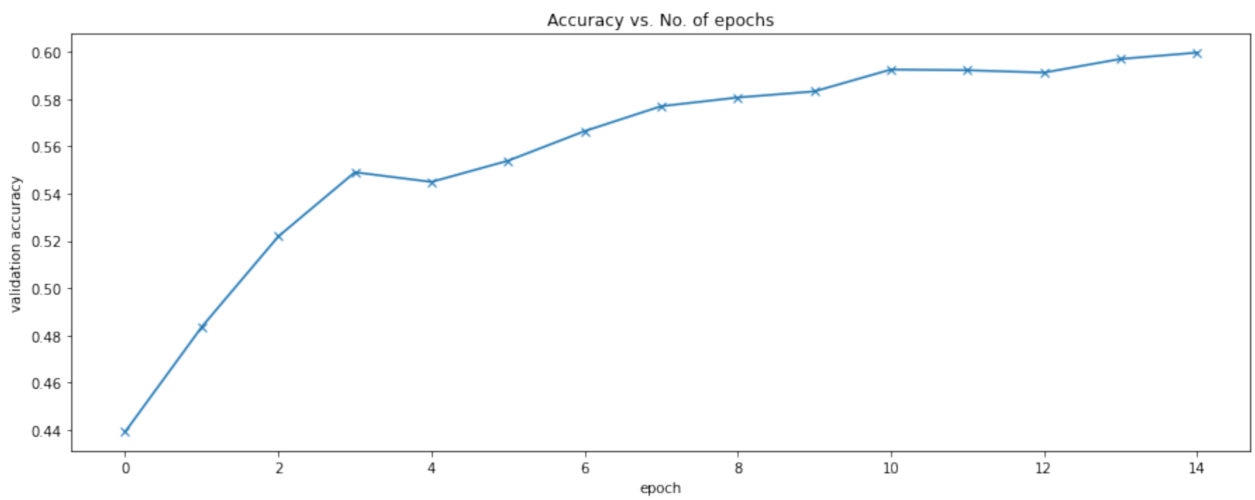

```
GroundTruth:    cat  ship  ship plane
```

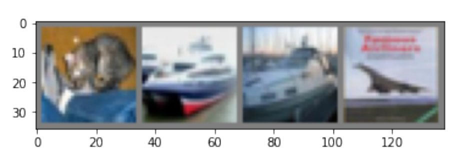
```
Accuracy:  63.64%
```

2.2.3 Using Batch normalization and Leaky_Relu activation functions

```
Epoch [0], train_loss: 1.6639, val_loss: 1.5182, val_acc: 0.4392
Epoch [1], train_loss: 1.4383, val_loss: 1.4117, val_acc: 0.4836
Epoch [2], train_loss: 1.3537, val_loss: 1.3204, val_acc: 0.5219
Epoch [3], train_loss: 1.2949, val_loss: 1.2780, val_acc: 0.5491
Epoch [4], train_loss: 1.2569, val_loss: 1.2776, val_acc: 0.5450
Epoch [5], train_loss: 1.2269, val_loss: 1.2515, val_acc: 0.5539
Epoch [6], train_loss: 1.1909, val_loss: 1.2202, val_acc: 0.5665
Epoch [7], train_loss: 1.1788, val_loss: 1.1920, val_acc: 0.5770
Epoch [8], train_loss: 1.1538, val_loss: 1.1824, val_acc: 0.5807
Epoch [9], train_loss: 1.1398, val_loss: 1.1773, val_acc: 0.5833
Epoch [10], train_loss: 1.1294, val_loss: 1.1540, val_acc: 0.5926
Epoch [11], train_loss: 1.1067, val_loss: 1.1678, val_acc: 0.5923
Epoch [12], train_loss: 1.1011, val_loss: 1.1608, val_acc: 0.5913
Epoch [13], train_loss: 1.0888, val_loss: 1.1391, val_acc: 0.5970
Epoch [14], train_loss: 1.0772, val_loss: 1.1194, val_acc: 0.5997
```





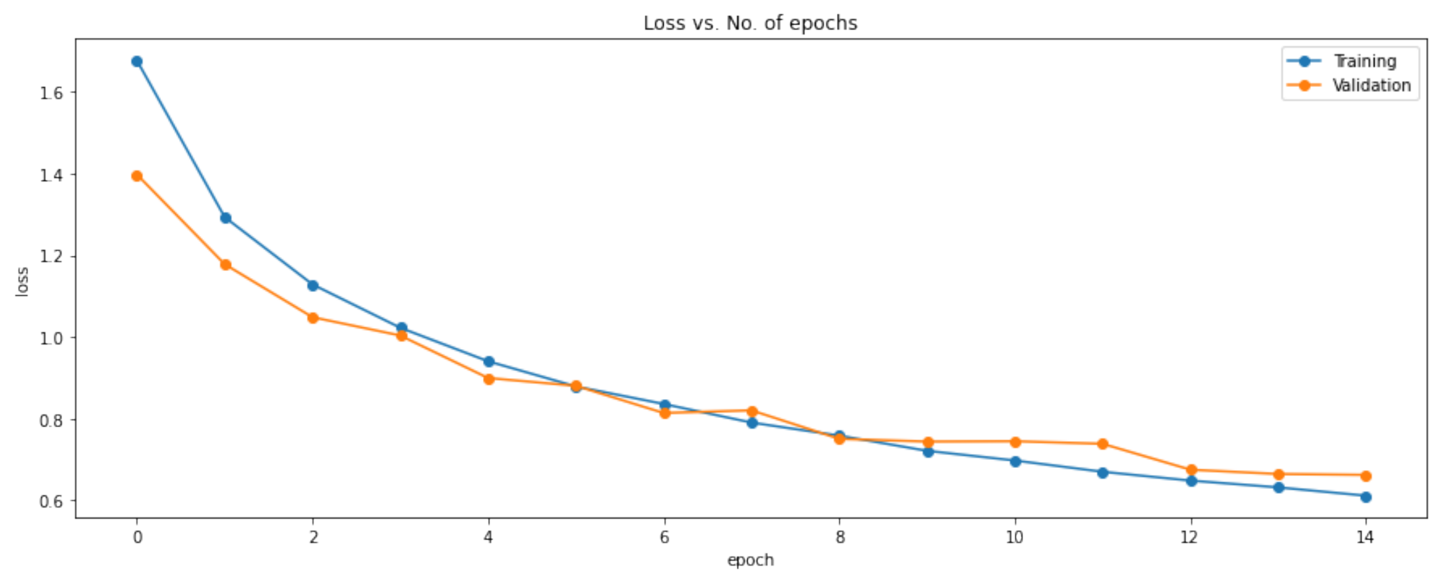


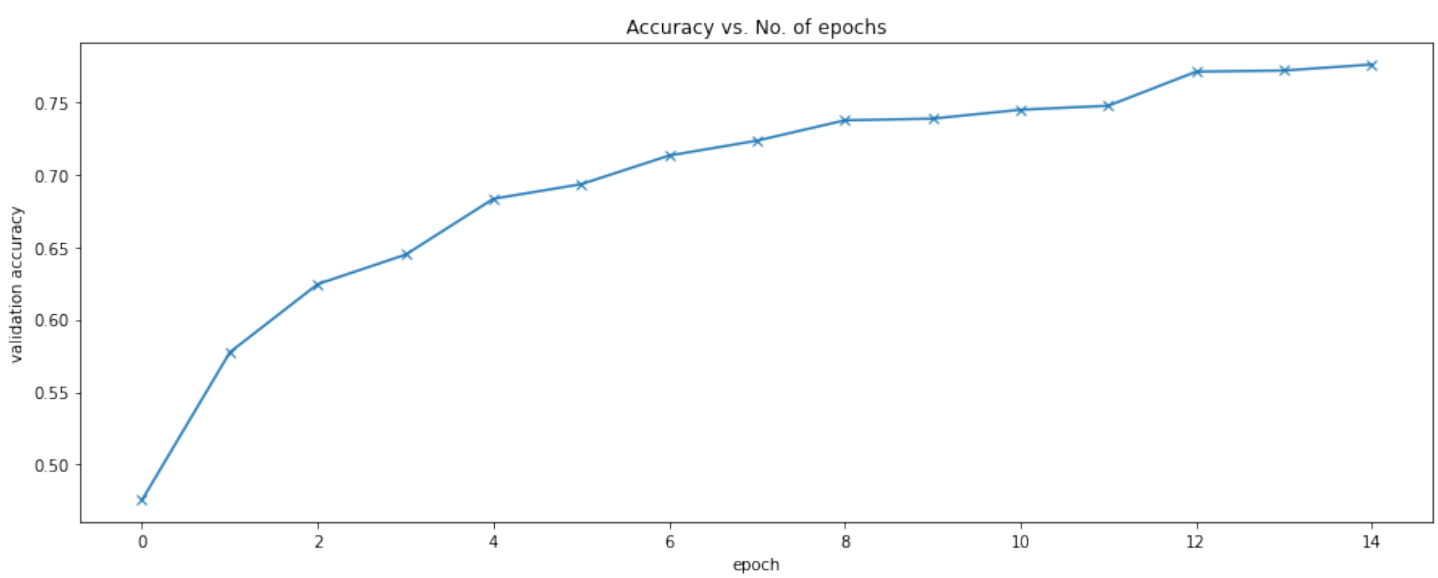
```
GroundTruth:    cat  ship  ship plane
```

2.2.4 Optimal parameters

In comparison Adam optimizer and learning rate = 0.001 produces much better results than SGD. Moreover, adding convolutional layers, dropout and batch normalization improves accuracy significantly as it can be appreciated in the following results:

```
Epoch [0], train_loss: 1.6764, val_loss: 1.3976, val_acc: 0.4755
Epoch [1], train_loss: 1.2933, val_loss: 1.1780, val_acc: 0.5775
Epoch [2], train_loss: 1.1287, val_loss: 1.0484, val_acc: 0.6246
Epoch [3], train_loss: 1.0226, val_loss: 1.0036, val_acc: 0.6450
Epoch [4], train_loss: 0.9406, val_loss: 0.8996, val_acc: 0.6835
Epoch [5], train_loss: 0.8790, val_loss: 0.8807, val_acc: 0.6936
Epoch [6], train_loss: 0.8361, val_loss: 0.8140, val_acc: 0.7134
Epoch [7], train_loss: 0.7906, val_loss: 0.8205, val_acc: 0.7237
Epoch [8], train_loss: 0.7587, val_loss: 0.7507, val_acc: 0.7377
Epoch [9], train_loss: 0.7215, val_loss: 0.7441, val_acc: 0.7389
Epoch [10], train_loss: 0.6978, val_loss: 0.7449, val_acc: 0.7451
Epoch [11], train_loss: 0.6702, val_loss: 0.7387, val_acc: 0.7478
Epoch [12], train_loss: 0.6487, val_loss: 0.6753, val_acc: 0.7714
Epoch [13], train_loss: 0.6320, val_loss: 0.6648, val_acc: 0.7722
Epoch [14], train_loss: 0.6117, val_loss: 0.6626, val_acc: 0.7763
```





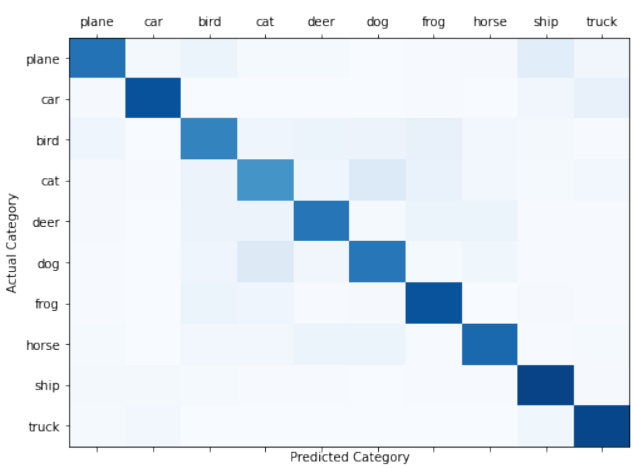

```
GroundTruth:   ship   dog truck  ship
```

```
Overall Accuracy:  78.54%
```

Top-1 prediction accuracy:

```
Accuracy of plane : 83 %
Accuracy of   car : 88 %
Accuracy of  bird : 65 %
Accuracy of   cat : 58 %
Accuracy of  deer : 65 %
Accuracy of   dog : 67 %
Accuracy of  frog : 85 %
Accuracy of horse : 71 %
Accuracy of  ship : 94 %
Accuracy of truck : 89 %
```

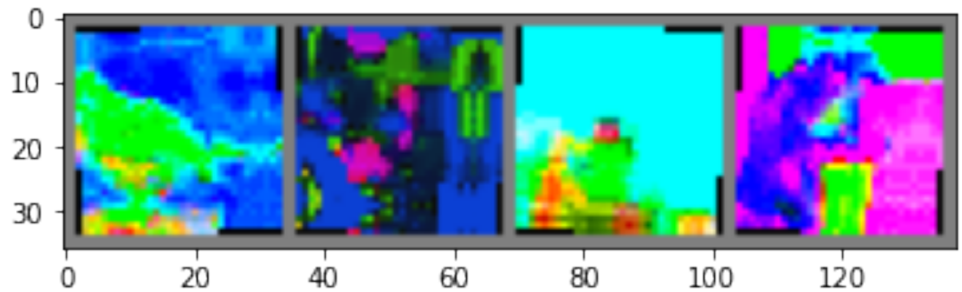
```
Confusion matrix
```




```
[746  19  53  12  13   0   5   8 109  35] (0) plane
[  9 871   2   1   1   1   5   1  32  77] (1) car
[ 46   2 677  43  52  57  76  25  17   5] (2) bird
[ 10   6  56 615  44 135  67  28  15  24] (3) cat
[ 10   0  56  55 733  14  61  61   5   5] (4) deer
[  7   2  41 132  34 726  12  36   5   5] (5) dog
[  6   1  52  41   7  10 867   1  11   4] (6) frog
[ 14   2  27  30  59  60   5 784   7  12] (7) horse
[ 16  16  12   5   4   3   7   4 922  11] (8) ship
[ 14  25   1   3   1   2   3   2  36 913]] (9) truck
```

2.3 Example of failed cases

Changing drastically the training data with transformations such as a different hue space or saturation values or random rotations gives above 50% accuracy which shows the strength of Convolutional Neural Networks to work well any type data initialization and regularization.



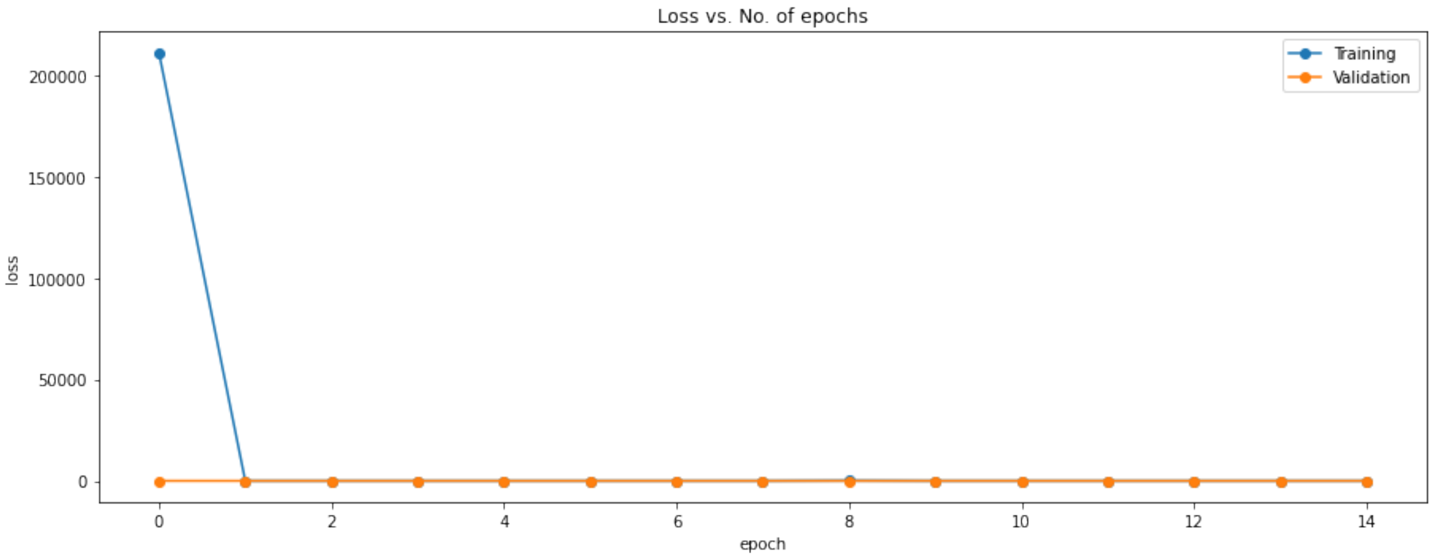
GroundTruth: bird   cat ship   cat

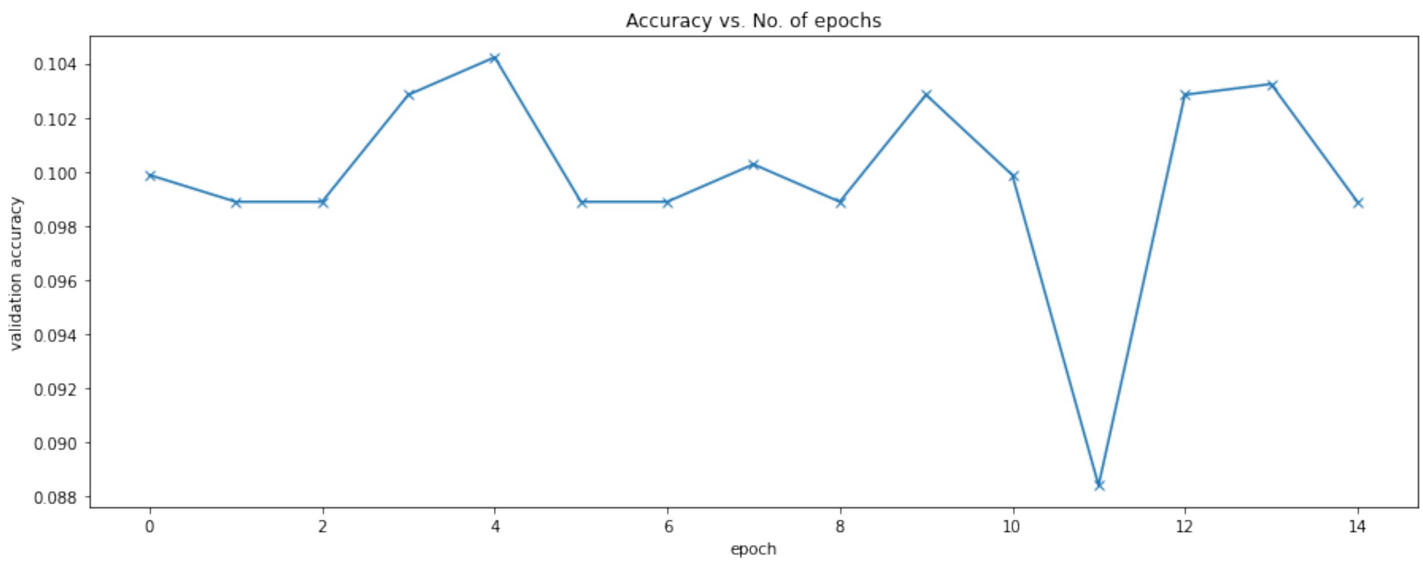
More important are hyperparameters such as learning rate which can affect learning significantly if the wrong value is chosen.

2.3.1 High learning rate = Fail

As seen in class a low learning rate will take a long time to learn where as a high learning rate can make the gradient either to explode or to jump around without failing in local minima which is what occurs in the following example:

```
Epoch [0], train_loss: 211266.4182, val_loss: 2.3638, val_acc: 0.0999
Epoch [1], train_loss: 2.3686, val_loss: 2.4382, val_acc: 0.0989
Epoch [2], train_loss: 2.3774, val_loss: 2.3666, val_acc: 0.0989
Epoch [3], train_loss: 2.3712, val_loss: 2.3770, val_acc: 0.1028
Epoch [4], train_loss: 2.3667, val_loss: 2.4131, val_acc: 0.1042
Epoch [5], train_loss: 2.3759, val_loss: 2.3973, val_acc: 0.0989
Epoch [6], train_loss: 2.3689, val_loss: 2.3469, val_acc: 0.0989
Epoch [7], train_loss: 2.3724, val_loss: 2.3493, val_acc: 0.1003
Epoch [8], train_loss: 118.2178, val_loss: 2.3806, val_acc: 0.0989
Epoch [9], train_loss: 2.3677, val_loss: 2.4584, val_acc: 0.1028
Epoch [10], train_loss: 2.3771, val_loss: 2.3687, val_acc: 0.0999
Epoch [11], train_loss: 2.3705, val_loss: 2.4444, val_acc: 0.0884
Epoch [12], train_loss: 2.3813, val_loss: 2.5112, val_acc: 0.1028
Epoch [13], train_loss: 2.3778, val_loss: 2.4851, val_acc: 0.1032
Epoch [14], train_loss: 2.3776, val_loss: 2.3423, val_acc: 0.0989
```





This model makes all test cases to fall under one category. It is straightforward to see that it would reach only around 10% accuracy which demonstrates why is important to assign an appropriate learning rate that does not underfit (high learning rate) nor overfit (low learning rate) our model.


```
GroundTruth:    truck  truck  truck  truck
```