HACETTEPE
ÜNİVERSİTESİ

**Hacettepe University - Ankara**

# Programming Assignment 3
# BBM418 Report

Written by:

**Berk Karaimer -b21827541**
Departement of Computer Science

**Academic Year: 2020-2021**

**Dataset creation**

Division between train, test and validation was done as 60% train, 20% test and 20% validation from all available images in the dataset that was given to us. I separated them manually into folders as train, test and validation and within the folders, I added 15 folders for each category and added the dataset for them as mentioned.

# Part 1

Firstly, the first cnn without residual connections is called as following;

In [23]:
```
num_epochs = 100
opt_func = torch.optim.Adam
lr = 0.002
model = SceneClassification()
if torch.cuda.is_available():
    model.cuda()

#fitting the model on training data and record the result after each epoch
history = fit(num_epochs, lr, model, train_dl, val_dl, opt_func)
```

Epochs were given to us as 100, learning rate is set from here as lr and optimization function is set here as well. From here, the fit function is called and inside of it, the model starts the training. In the same block, evaluation and accuracy is calculated as well. Accuracy is calculated as correct predictions divided by all predictions. Evaluation function calls for validation_step function within ImageClassificationBase class and calculates loss by cross_entropy.

Both the models has 5 convolution layers and to keep the images size same as input, the kernel size and paddings are set as following. In channel is set as 3 since the image has 3 dimensions as RBG. I used resizing to dataset as 32 by 32 as creation of dataset within code.

In [20]:
```
class SceneClassification(ImageClassificationBase):
    def __init__(self):
        super().__init__()
        self.network = nn.Sequential(

            nn.Conv2d(3, 4, kernel_size = 3, padding = 1),
            nn.ReLU(),
            nn.Conv2d(4,4, kernel_size = 3, stride = 1, padding = 1),
            nn.ReLU(),
            nn.MaxPool2d(2,2),

            nn.Conv2d(4, 16, kernel_size = 3, stride = 1, padding = 1),
            nn.ReLU(),
            nn.Conv2d(16 ,16, kernel_size = 3, stride = 1, padding = 1),
            nn.ReLU(),
            nn.MaxPool2d(2,2),

            nn.Conv2d(16, 32, kernel_size = 3, stride = 1, padding = 1),
            nn.ReLU(),
            nn.MaxPool2d(2,2),

            nn.Flatten(),
            nn.Linear(512,128),
            nn.ReLU(),
            nn.Linear(128,15)
        )

    def forward(self, xb):
        return self.network(xb)
```

Convolutions are firstly done as two block with each one containing 2 convolution, two ReLU and one maxpooling. Out channels are set as powers of two and low as possible since it was taking too much memory of GPU. Fully connected layers input value is set by calculating the outputs of maxpools(32 from conv layers out, 16 from pooling). Within the FC I reduced the output by ¼ and the final fc layers output is set as class size which is 15.

For implementation of residual network, I created a new class as ResNet. Within the class, the layers are created as can be seen. 5 layers are created as first two giving 16 out channels, next 2 giving 32 and last one with 64.

```
In [26]: # 3x3 convolution
         def conv3x3(in_channels, out_channels, stride=1):
             return nn.Conv2d(in_channels, out_channels, kernel_size=3,
                              stride=stride, padding=1, bias=False)
```

Stride is set by 1 if not given any parameters and padding is set as 1. With layer 4 and 5 giving stride as 2, the image output is reduced to 8 and with pooling it drops to 1 (64x1x1) and the fully connected layers first in channel is set as 64 and out is ¼ again. Second FC out channel is set to classes size.

The following page contains the code for ResNet class.

In [28]:
```python
# ResNet
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=15):
        super(ResNet, self).__init__()
        self.in_channels = 16
        self.conv = conv3x3(3, 16)
        self.bn = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self.make_layer(block, 16, layers[0])
        self.layer2 = self.make_layer(block, 16, layers[0])
        self.layer3 = self.make_layer(block, 32, layers[0])
        self.layer4 = self.make_layer(block, 32, layers[1], 2)
        self.layer5 = self.make_layer(block, 64, layers[2], 2)
        self.avg_pool = nn.AvgPool2d(8)
        self.fc = nn.Linear(64, 16)
        self.dropout = nn.Dropout(0.2)
        self.fc2 = nn.Linear(16, num_classes)
        self.dropout = nn.Dropout(0.2)

    def make_layer(self, block, out_channels, blocks, stride=1):
        downsample = None
        if (stride != 1) or (self.in_channels != out_channels):
            downsample = nn.Sequential(
                conv3x3(self.in_channels, out_channels, stride=stride),
                nn.BatchNorm2d(out_channels))
        layers = []
        layers.append(block(self.in_channels, out_channels, stride, downsample))
        self.in_channels = out_channels
        for i in range(1, blocks):
            layers.append(block(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv(x)
        out = self.bn(out)
        out = self.relu(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.layer5(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        #out = self.fc(out)
        out = self.dropout(self.fc(out))
        #out = self.fc2(out)
        out = self.dropout(self.fc2(out))
        return out
```

## 0.1 Training and Evaluating your model

Loss is calculated by cross entropy and accuracy is calculated by dividing the correct labels by all the labels from tested data.

First model uses ImageClassificationBase class and calculates the loss within the class.

In [19]:
```
import torch.nn as nn
import torch.nn.functional as F

class ImageClassificationBase(nn.Module):

    def training_step(self, batch):
        images, labels = batch
        images, labels = images.cuda(), labels.cuda()
        out = self(images)                  # Generate predictions
        loss = F.cross_entropy(out, labels) # Calculate loss
        return loss

    def validation_step(self, batch):
        images, labels = batch
        images, labels = images.cuda(), labels.cuda()
        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}
```

Second model calculates the loss as can be seen from the following snippet.

In [30]:
```
model = ResNet(ResidualBlock, [2, 2, 2]).cuda()

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
```

Accuracy of the first model is calculated as following, which divides correct predictions by all predictions.

In [22]:
```
def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))
```

Second model's accuracy is calculated as following which is same.

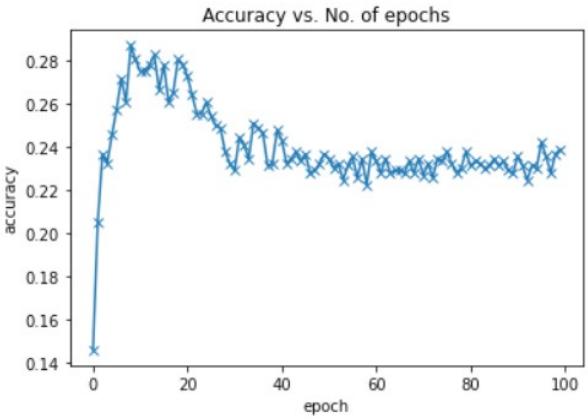
In [33]:
```
#Test the model
model.eval()
accuracy = []
test_predicts = []
test_labelss = []
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_dl:
        images, labels = images.cuda(), labels.cuda()
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        accuracy.append(correct / total)
```

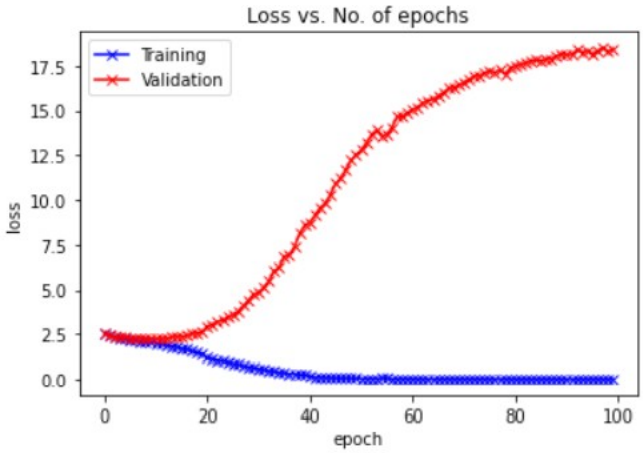
**1. Draw a graph of loss and accuracy change**

Batch size = 8

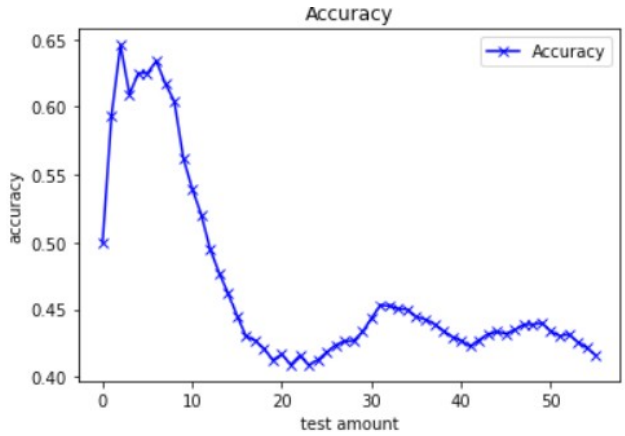
Lr = 0.002 (with epoch % 20 , lr is divided by 2)

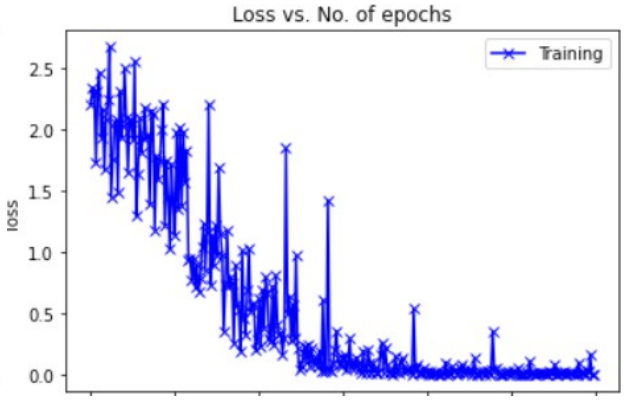
First model:




Second model:

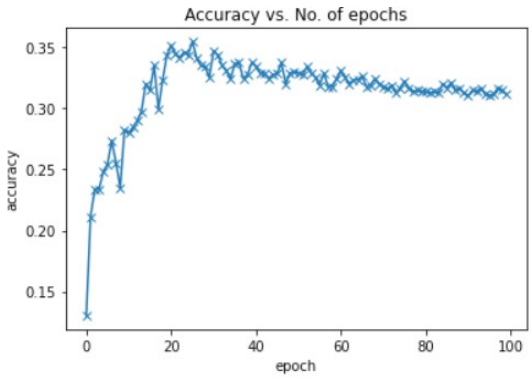


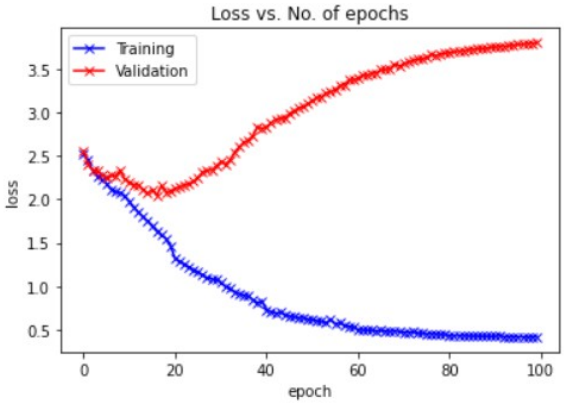



Batch size = 16

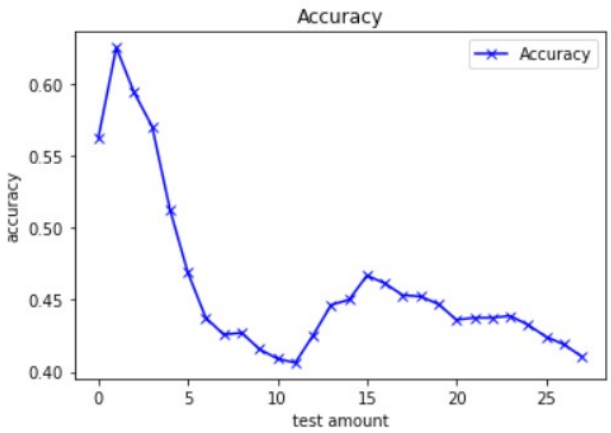
Lr = 0.001 (with epoch % 20 , lr is divided by 3)

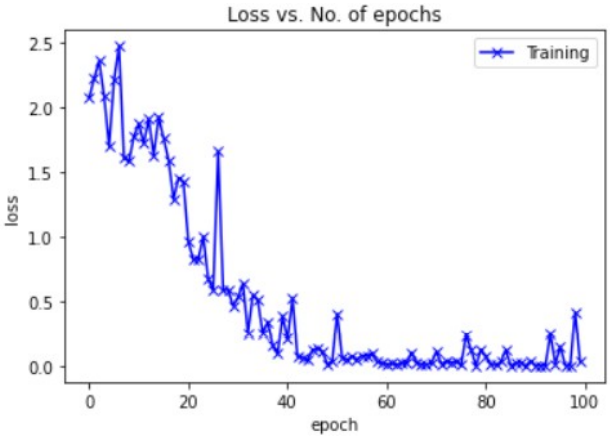
First model:




Second model:

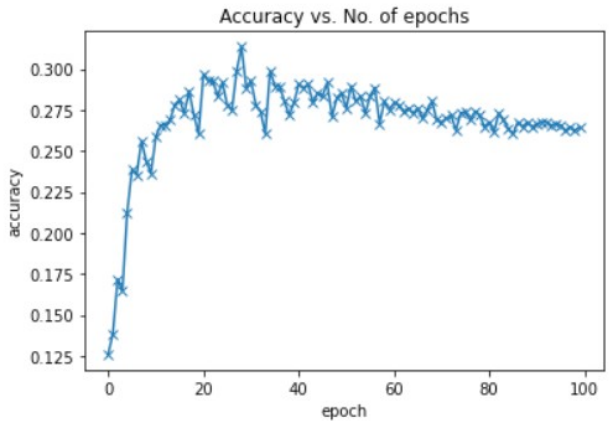


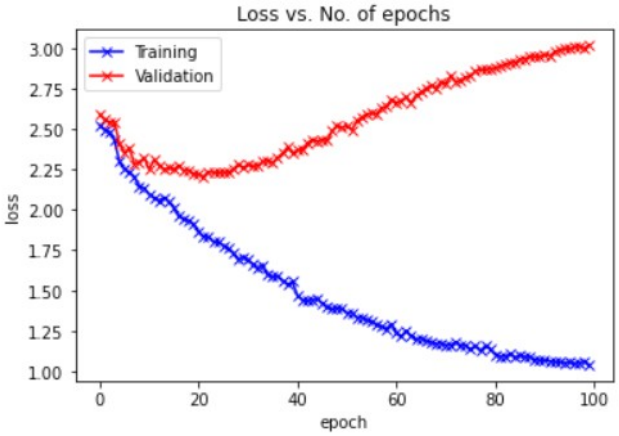



Batch size = 32

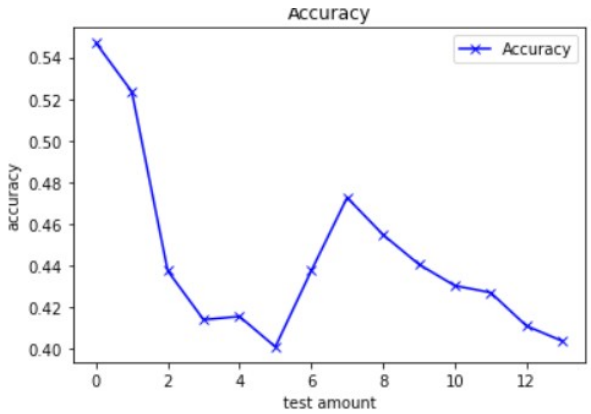
Lr = 0.001 (with epoch % 20 , lr is divided by 2)

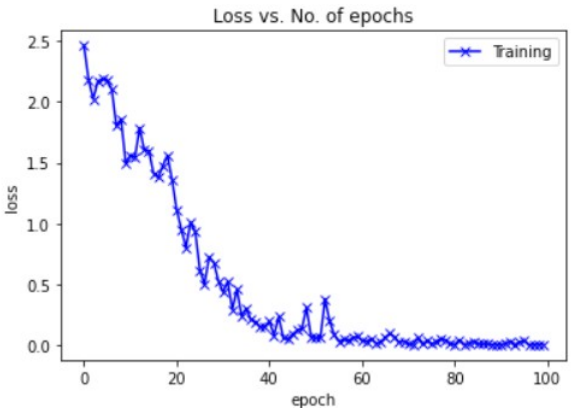
First model:




Second model:

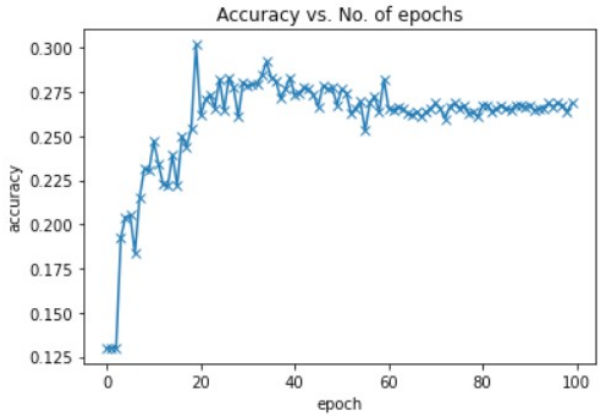


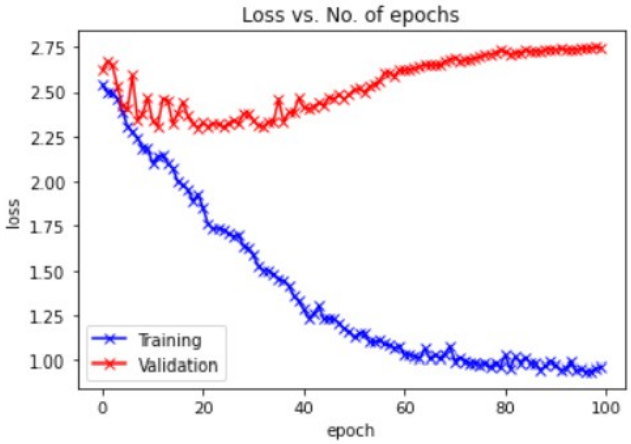



Batch size = 64

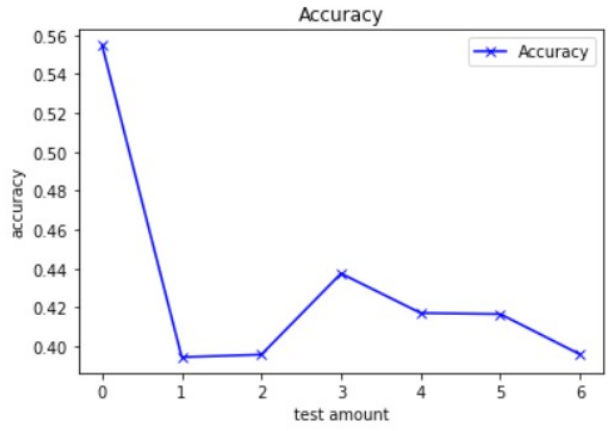
Lr = 0.002 (with epoch % 20 , lr is divided by 3)

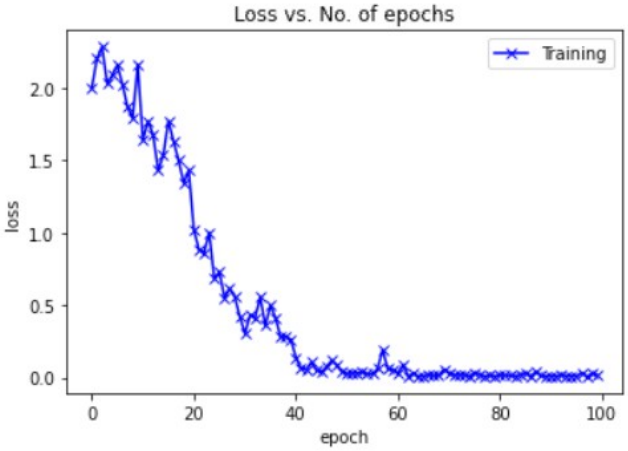
First model:




Second model:



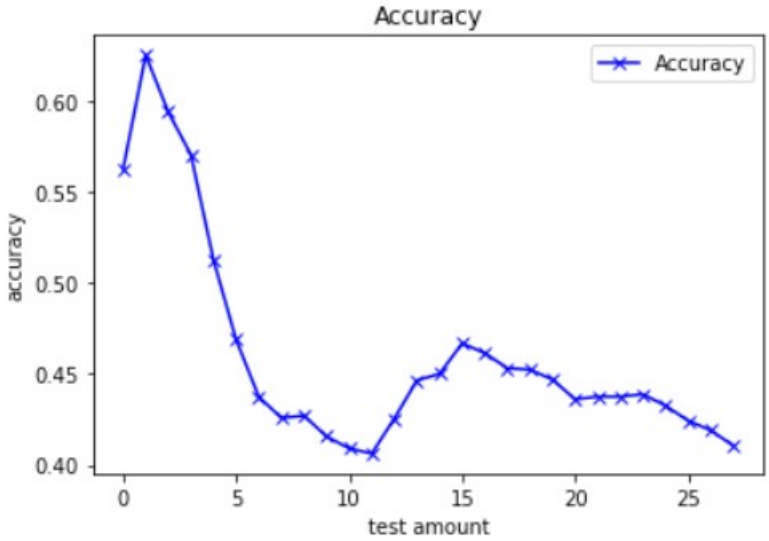



**2. Select your best model with respect to validation accuracy and give test accuracy result.**

Best model that I had was ResNet with batch size of 16 and Lr as 0.001 which was divided by 3 gradually.

Test result of the resnet model for this case was as following;

Accuracy of the model on the test images: 41.05145413870246 %

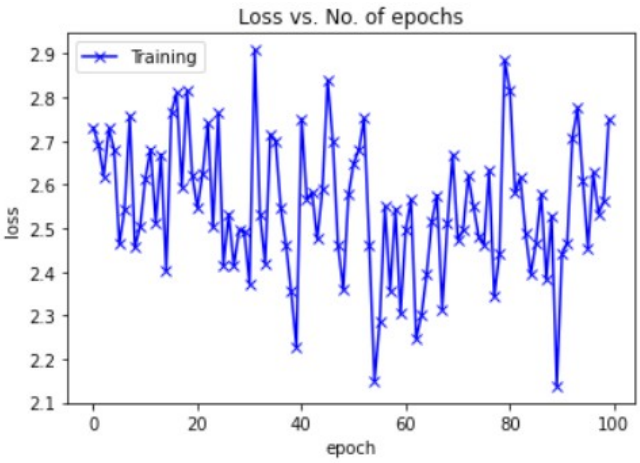


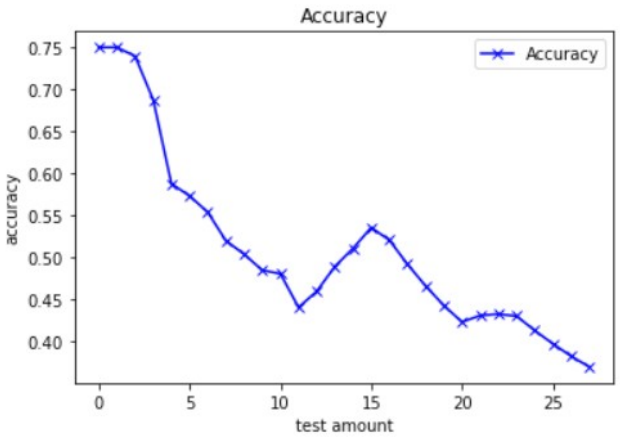

**3. Integrate dropout to your best model**

As can be seen by the previously given ResNet implementation snippet, since it performed better than rest, I added dropouts to that model.

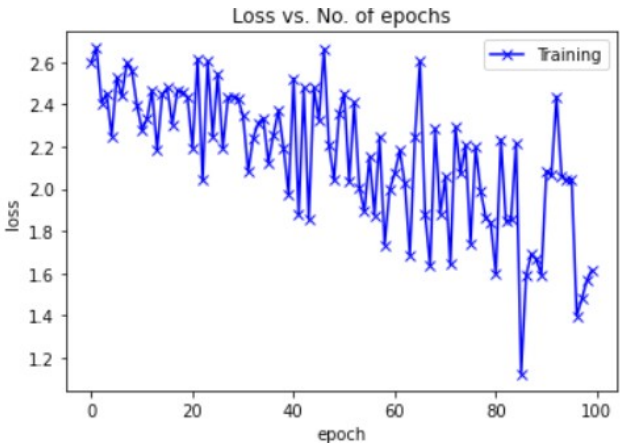
I added dropouts for 2 FC layers since this was the proposed way of using the dropouts by the creator.

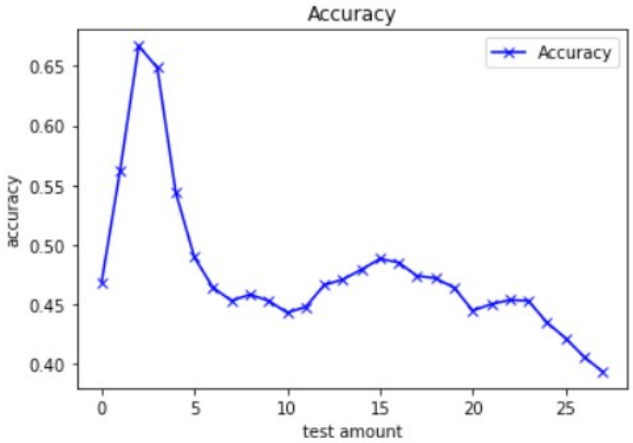
Dropout = 0.7 for ResNet model



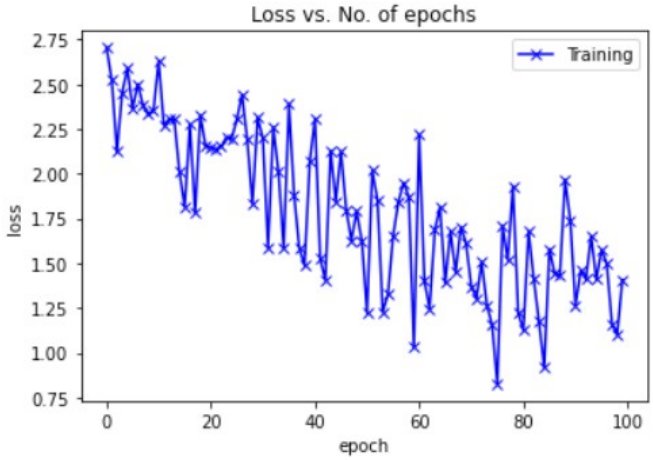



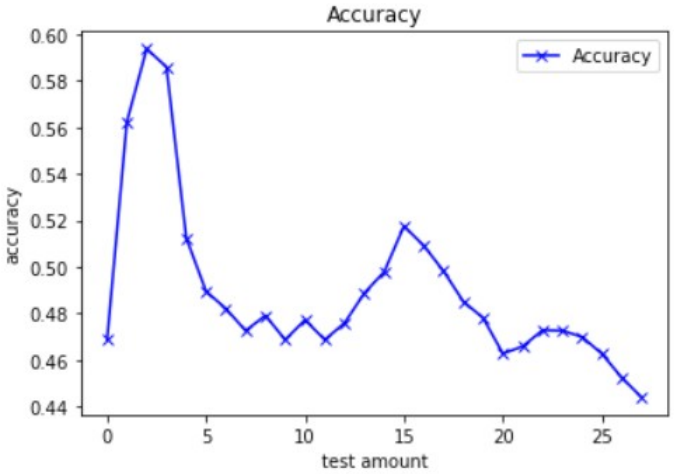
Dropout = 0.5 for ResNet model



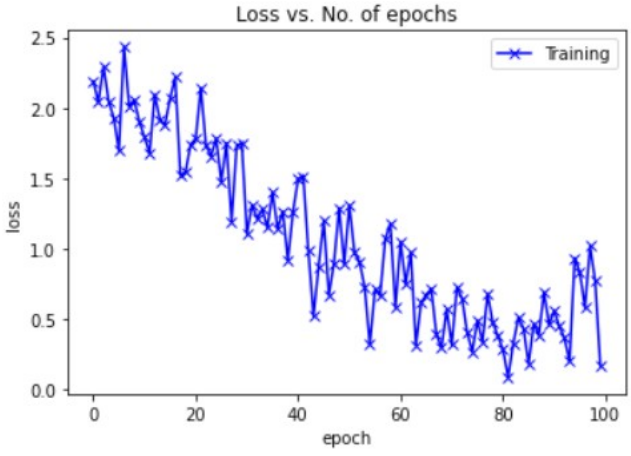



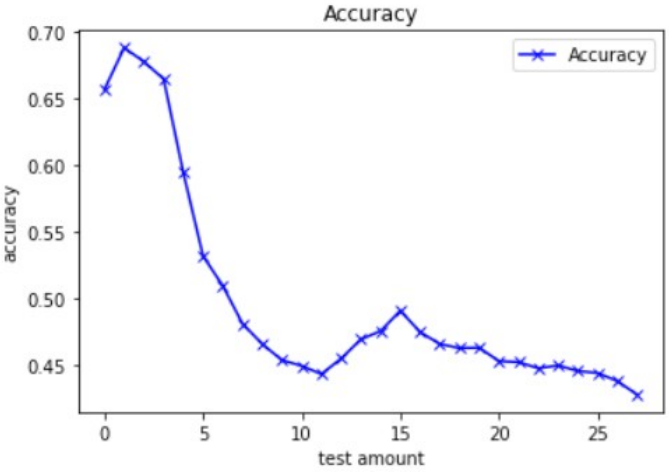

Dropout = 0.3 for Resnet model




Dropout = 0.2 for Resnet model

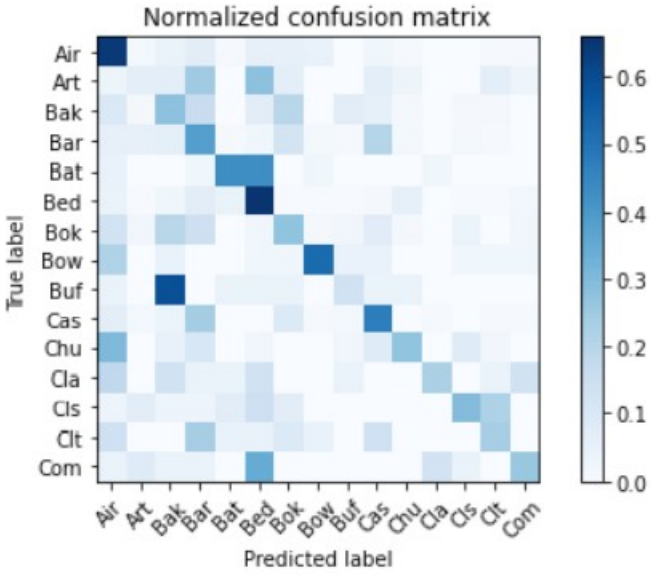


Best result I had from these Dropout values was for dropout set to 0.3.

Accuracy of the model on the test images: 44.407158836689035 %

**4. Plot a confusion matrix for your best model's predictions**




**5. Explain and analyze your findings and results**

From the results I got, the resnet model works slightly better for my implementation as can be seen from the result for accuracy I have gotten.

The dropouts with the resnet works different for the parameters that I have chosen, from the results for my model, high dropout values decreases the performance of the model and too low dropout results badly too.

Adding a dropout works better and increases the performance of the model, if the parameter is tested for the implemented model.

Learning rate value works similar with the dropout as well. Increasing it and decreasing it works different with given data. The value must be chosen by testing the model with learning rates and the performance with that learning rate for the model.

Batch sizes helps with the performance of the model since it helps with how much space on the GPU the program will have. It should be chosen by testing the model with different batch sizes and which one gives the best result as wanted for performance of the model and the machine that is being used.

Image size and the layer inputs outputs must be chosen carefully since they could easily overwhelm the machine that is running on. They have most affect on the performance results as well and are the most important part of the model.

# Part 2

## 1- Fine-tuning and explanations

Fine-tuning is the way of taking weights of a pre-trained neural network and use it to initialize a new model being trained on data.

This is used since it speeds up the training part of the model and it is helpful with small datasets. Since convolutional networks has larger parameters, with smaller dataset, this causes overfitting. If the dataset is not too different from the dataset that was used to train the model, the results would suffice.

We freeze the rest of the layers and only keep the Fully Connected layer since the output channel must be same value as our used dataset class's size.

To freeze only the FC layer, first, the layers parameters requires_grad is set to false for every layer as can be seen from following snippet.

```
In [54]: def set_parameter_requires_grad(model, feature_extracting):
             if feature_extracting:
                 for param in model.parameters():
                     param.requires_grad = False # make only the fc layer learnable
```

Next, the layer that was not frozen is set as the following snippet. I have set the in features as number of features from model and the out part is set as number of classes in our dataset. This way the only the fc layer is set by us.

```
In [57]: def initialize_model(model_name, num_classes, feature_extract, use_pretrained=True):
             # Initialize these variables which will be set in this if statement. Each of these
             #   variables is model specific.
             model_ft = None
             input_size = 0

             if model_name == "resnet":
                 """ Resnet18
                 """
                 model_ft = models.resnet18(pretrained=use_pretrained)
                 set_parameter_requires_grad(model_ft, feature_extract)
                 num_ftrs = model_ft.fc.in_features
                 model_ft.fc = nn.Linear(num_ftrs, num_classes)
                 input_size = 224
             else:
                 print("Invalid model name, exiting...")
                 exit()
             return model_ft, input_size
```

Dataloaders and the dataset it transformed for the size the pre-trained model wants, which is 224 for input size of resnet-18. And the model is initialized.

```
In [58]: # Initialize the model for this run
         model_ft, input_size = initialize_model(model_name, num_classes, feature_extract, use_pretrained=True)

         # Print the model we just instantiated
         print(model_ft)
```

The editted layer can be seen from the output of print(model_ft) as following snip.

```
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=15, bias=True)
)
```

Finally, the models unfrozen layers parameters has to be updated and the optimizer must be set as the following snip shows.

In [61]:
```
# Send the model to GPU
model_ft = model_ft.to(device)

# Gather the parameters to be optimized/updated in this run. If we are
#  finetuning we will be updating all parameters. However, if we are
#  doing feature extract method, we will only update the parameters
#  that we have just initialized, i.e. the parameters with requires_grad
#  is True.
params_to_update = model_ft.parameters()
print("Params to learn:")
if feature_extract:
    params_to_update = []
    for name,param in model_ft.named_parameters():
        if param.requires_grad == True:
            params_to_update.append(param)
            print("\t",name)
else:
    for name,param in model_ft.named_parameters():
        if param.requires_grad == True:
            print("\t",name)

# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(params_to_update, lr=0.001, momentum=0.9)
```

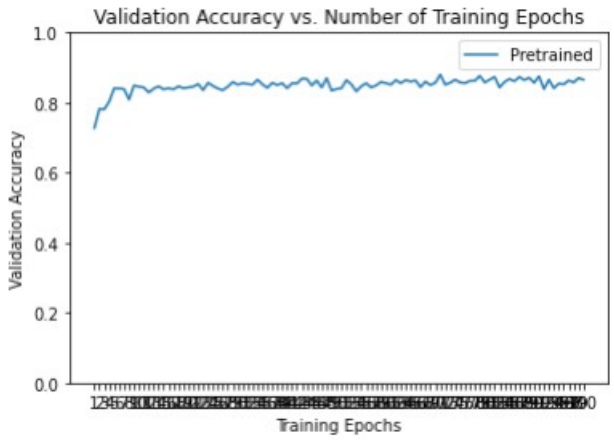
```
Params to learn:
         fc.weight
         fc.bias
```

After this part, the loss function is set and training, evaluation is done for given epoch size.

## 2.1- Train FC layer and freeze rest

This model was shown with snippet in the previous section, so I will be showing the results of this model only in this section.

Validation accuracy for the first model with only FC layers unfrozen is as following




Test result for this model is : Accuracy of the model on the test images: 83.668903803132 %

## 2.2- Train FC layer and Last Layer, Freeze Rest

Initializing of this model is invoked as following snip.

```
In [68]: # Initialize the model for this run
         model_ft2, input_size = initialize_model2(model_name, num_classes, feature_extract, use_pretrained=True)

         # Print the model we just instantiated
         print(model_ft2)
```

Layers of this model are frozen as the following snip.

```
In [66]: def set_parameter_requires_grad2(model):
             child_counter = 0
             for child in model.children():
                 if child_counter < 7:
                     print("child ",child_counter," was frozen")
                     for param in child.parameters():
                         param.requires_grad = False
                 elif child_counter == 7:
                         print("child ",child_counter," was not frozen")
                 elif child_counter == 8:
                     print("child ",child_counter," was frozen")
                     for param in child.parameters():
                         param.requires_grad = False
                 else:
                     print("child ",child_counter," was not frozen")
                 child_counter += 1
```

Output of this showing which layers were frozen can be seen by the following snip and the layers are shown as well. I have changed the layer 4 as following which is child 7 and fc as following which is child 9.

```
child  0  was frozen
child  1  was frozen
child  2  was frozen
child  3  was frozen
child  4  was frozen
child  5  was frozen
child  6  was frozen
child  7  was not frozen
child  8  was frozen
child  9  was not frozen
```

```
  )
  (layer4): Sequential(
    (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
```

ıks/CV_HW3.ipynb

CV_HW3 - Jupyter Notebook

```
  (fc): Linear(in_features=512, out_features=15, bias=True)
)
```

This is done in initializing function as can be seen from following snip.

In [67]:
```
def initialize_model2(model_name, num_classes, feature_extract, use_pretrained=True):
    # Initialize these variables which will be set in this if statement. Each of these
    #   variables is model specific.
    model_ft = None
    input_size = 0

    if model_name == "resnet":
        """ Resnet18
        """
        model_ft = models.resnet18(pretrained=use_pretrained)
        set_parameter_requires_grad2(model_ft)

        model_ft.layer4 = nn.Sequential(nn.Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), b
                                        nn.BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
                                        nn.ReLU(inplace=True),
                                        nn.Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), b
                                        nn.BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
        num_ftrs = model_ft.fc.in_features
        model_ft.fc = nn.Linear(num_ftrs, num_classes)
        input_size = 224
    else:
        print("Invalid model name, exiting...")
        exit()
    return model_ft, input_size
```

In this part to modify the layer4 I have removed some layers and changed them with these layers which is less than original layer.

Parameters for this and the optimization is done as the following snip shows:

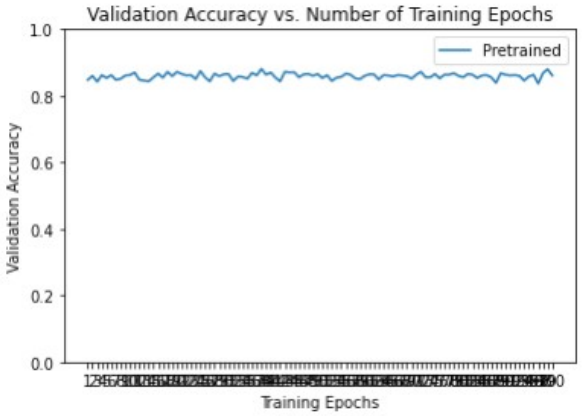
In [69]:
```
# Send the model to GPU
model_ft2 = model_ft.to(device)

# Gather the parameters to be optimized/updated in this run. If we are
#  finetuning we will be updating all parameters. However, if we are
#  doing feature extract method, we will only update the parameters
#  that we have just initialized, i.e. the parameters with requires_grad
#  is True.
params_to_update = model_ft2.parameters()
print("Params to learn:")
if feature_extract:
    params_to_update = []
    for name,param in model_ft2.named_parameters():
        if param.requires_grad == True:
            params_to_update.append(param)
            print("\t",name)
else:
    for name,param in model_ft2.named_parameters():
        if param.requires_grad == True:
            print("\t",name)

# Observe that all parameters are being optimized
optimizer_ft2 = optim.SGD(params_to_update, lr=0.001, momentum=0.9)
```

```
Params to learn:
         fc.weight
         fc.bias
```

and the model is train as the following snip:

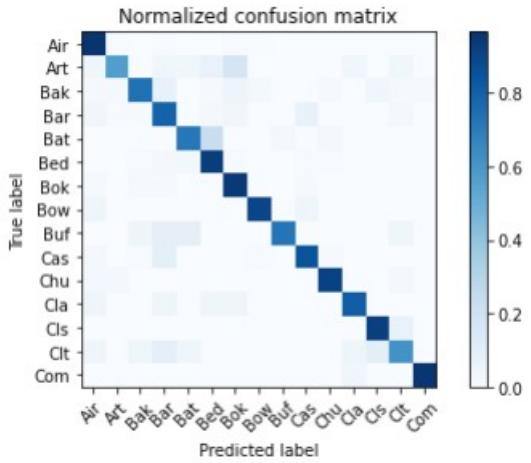
In [70]:
```
# Setup the loss fxn
criterion = nn.CrossEntropyLoss()

# Train and evaluate
model_ft2, hist = train_model(model_ft2, dataloaders_dict, criterion, optimizer_ft2, num_epochs=num_epochs, is_i
```

Validation accuracy for this model can be seen from following histogram and the test accuracy result of this is model is: Accuracy of the model on the test images: 85.57046979865771 %

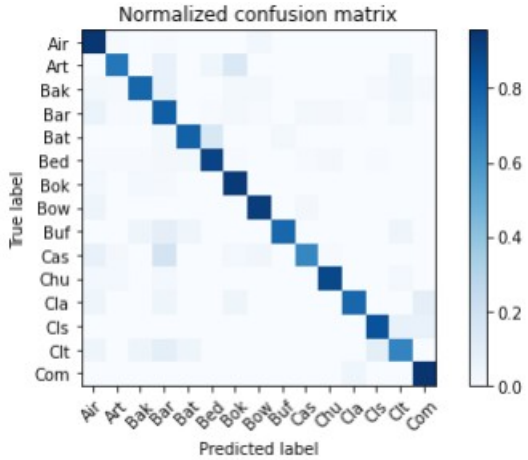


## 3. Plot confusion matrix for your best model and analyze results

Confusion matrix of second model which gave the best result:




and first model:




As far as I can see from these results, some classes such as art for first model is more accurate, but in the second model which is the first matrix given, the negative results are less such as bar class.

From the test results the second model which is first matrix has better accuracy but the are not too far off as can be seen from these matrices. Still changing some layers in the model seems to have worked better from these results.

## 4. Compare and analyze your results in Part-1 and Part-2.

The results of the part-1 was at most ~45% and the part-2 was at most ~85% which is nearly 100% better performance. This shows that pre-trained models works really well if their trained dataset was similar to the ones we are using.

From these results, I can say that models that I have made by testing their parameters and trying to get better results was not performing well. This is most likely to be setting learning rate too low from what it should be for a model that was not pre-trained and the layers not setting well to perform well. Layer's parameters are another reason for this result. The layers I have created are way too simple considering layers of the pre-trained model. Image sizes were resized way too low to keep the memory of gpu less occupied which made the result considerably worse from what it could be. I have set the convolution layers outputs low for the same reason.

To make these models in part-1, I tried setting the learning rate higher than what it is for part-2 models and at first I thought that the 45% accuracy was good for my first try. After seeing the result of the part-2 I have realized my mistakes for part-1 but still satisfied enough.