Deep Learning Homework 2

Student ID: 0416106 **Name**: 彭敬樺

1. Convolutional Neural Network for Image Recognition

i. Preprocessing stage is useful in this Image Recognition task due to the varying size of image, different aspect ratio, and the need for normalization.

For all data:

- Data is converted to tensor to be fed into the network
- Data is normalized to (-1, 1) to have better result with mean of 0(zero)
- Data that will be fed into the network is also separated into batches. In this case, batch size is 32.

For training data:

- Additional step is done before converting to tensor, that is using RandomResizedCrop function to resize the image. This is for more variety in the training data and has been proven to provide better result compared to standard resize
- Shuffle data that will be fed into the network to prevent bias

For validation data:

- Data that will be used is the same as training data, in exception of the function RandomResizedCrop. In validation step, normal Resize function is used instead.
- Data is also shuffled

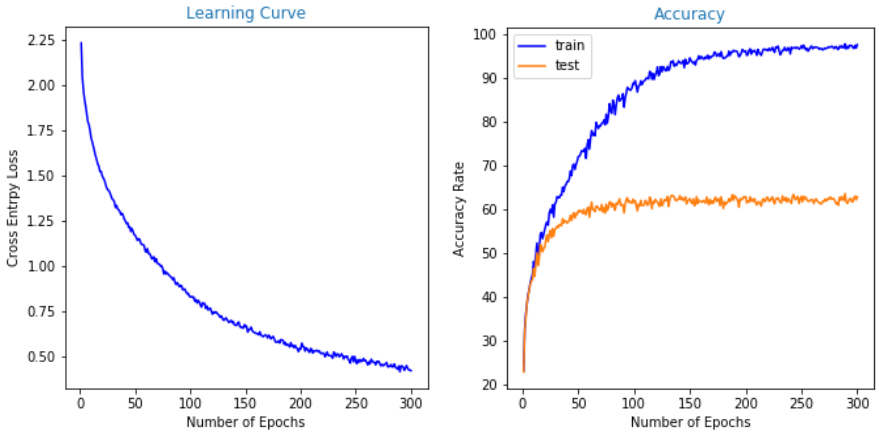
For testing data:

- Additional step is done before converting to tensor, that is using normal Resize function to resize the image.
- Data is not shuffled due to the lack of need when testing

In [3]:

```
train_data_path = 'animal-10/train/'
test_data_path = 'animal-10/val/'
BATCH_SIZE = 32
EPOCH = 300
INPUT_SIZE = 256

train_transform = transforms.Compose(
    [transforms.RandomResizedCrop((INPUT_SIZE,INPUT_SIZE)),
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

train_data = torchvision.datasets.ImageFolder(
    root = train_data_path,
    transform = train_transform
    )
trainloader = torch.utils.data.DataLoader(train_data, batch_size=BATCH_SIZE,
                                          shuffle=True, num_workers=2)

val_transform = transforms.Compose(
    [transforms.Resize((INPUT_SIZE,INPUT_SIZE)),
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

val_data = torchvision.datasets.ImageFolder(
    root = train_data_path,
    transform = val_transform
    )
valloader = torch.utils.data.DataLoader(val_data, batch_size=BATCH_SIZE,
                                          shuffle=True, num_workers=2)

test_transform = transforms.Compose(
    [transforms.Resize((INPUT_SIZE,INPUT_SIZE)),
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

test_data = torchvision.datasets.ImageFolder(
    root = test_data_path,
    transform = test_transform
    )
testloader = torch.utils.data.DataLoader(test_data, batch_size=BATCH_SIZE,
                                         shuffle=False, num_workers=2)

classes = ('butterfly', 'cat', 'chicken', 'cow', 'dog',
    'elephant', 'horse', 'sheep', 'spider', 'squirrel')
```

ii.



The network used in this portion is standard CNN, with configuration of layer:
- Convolution Layer of input filter of 3, output filter of 6, kernel size of 7, stride of 1, and padding of 3
- Max Pooling layer with 4x4 kernel size and stride of 4
- Relu activation function
- Convolutional Layer of input filter of 6, output filter of 16, kernel size of 5, stride of 1, and padding of 2
- Max Pooling layer with 2x2 kernel size and stride of 2
- Relu activation function
- Convolutional Layer of input filter of 16, output filter of 32, kernel size of 3, stride of 1, and padding of 1
- Max Pooling layer with 2x2 kernel size and stride of 2
- Relu activation function
- Fully Connected Layer with input 8192 nodes and output of 2048 nodes
- Relu activation function
- Fully Connected Layer with input 2048 nodes and output of 512 nodes
- Relu activation function
- Fully Connected Layer with input 512 nodes and output of 64 nodes
- Relu activation function
- Fully Connected Layer with input 64 nodes and output of 10 nodes
- Output of 10 nodes

In this scenario, padding don't really affect the testing accuracy due to the small proportion of padded value compared to the entire pixel count of the image (256x256 image). Stride of 2 and 3 increase training time but without significance improvement while having little worse result for testing accuracy. Next, bigger kernel size have its own advantage to be able to see in bigger view, which let it capture bigger context and object. Smaller kernel, on the other hand, handle subtle and smaller feature that exist in the input. By combining this 2 advantage and disadvantage, the result is quite satisfactory.

iii. The accuracy gain from butterfly is the highest, the most explainable reason is due to the feature that butterfly has that is most unique compared to the other animal. It could also be said that butterfly is significantly more different than the other category of animal. Some unique animal compared to the other class is elephant, spider and squirrel, which also has higher than average accuracy.

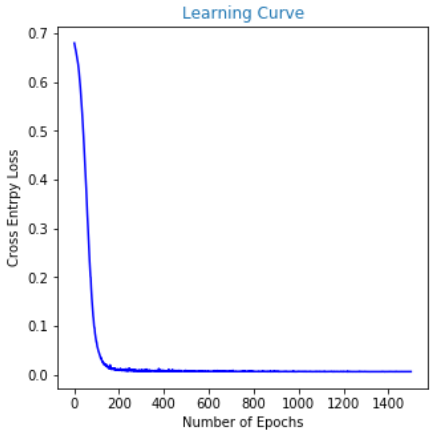
```
Accuracy of butterfly : 73 %
Accuracy of   cat : 51 %
Accuracy of chicken : 75 %
Accuracy of   cow : 50 %
Accuracy of   dog : 46 %
Accuracy of elephant : 75 %
Accuracy of horse : 68 %
Accuracy of sheep : 58 %
Accuracy of spider : 67 %
Accuracy of squirrel : 61 %
```

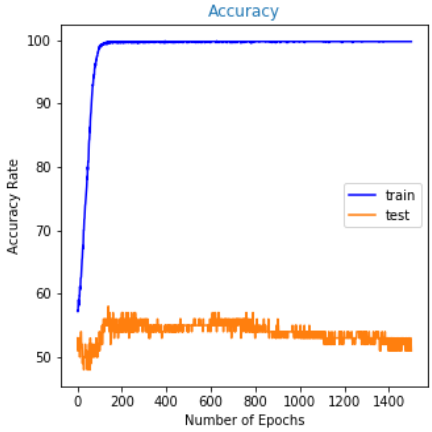
labels: butterfly, predict: butterfly

labels: dog, predict: cat

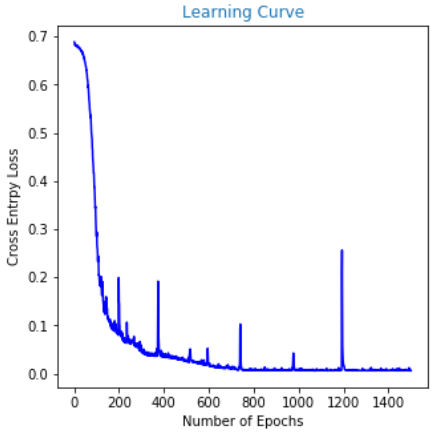
2. Recurrent Neural Network for Prediction of Paper Acceptance

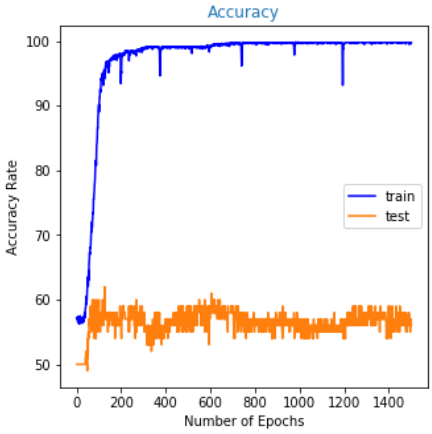
i. standard RNN





ii. LSTM





iii. The network for RNN and LSTM is very similar in this scenario, we have only change the standard memory into LSTM to gain the result shown above. In the LSTM scenario, there are more parameter to be trained, resulting in bigger network when saved in the memory. However, LSTM also has better result overall with higher peak and higher cliff compared to RNN. In this scenario, RNN result for testing accuracy keeps decreasing, this might happen due to overfitting in the training data, due to lack of training data. LSTM has more unstable graph for training accuracy, which show that overfitting occur, but due to the harsher resulting plane of LSTM, letting it to search larger space for more general result.