2021 - AI 825 - Visual Recognition
International Institute of Information Technology, Bangalore

Assignment 3

Shourabh Payal (MT2020054)

1. ***Convolution Neural Network***

   (a) **Experimentation and results**
   After experimenting with different medium sized networks, I obtained below results.
   Due to high compute time required, number of epochs were kept low while generating this plot. However experimentation was done with good number of epochs to get a fair idea.
   Please refer to final architecture section for better accuracy results in later sections of the report.

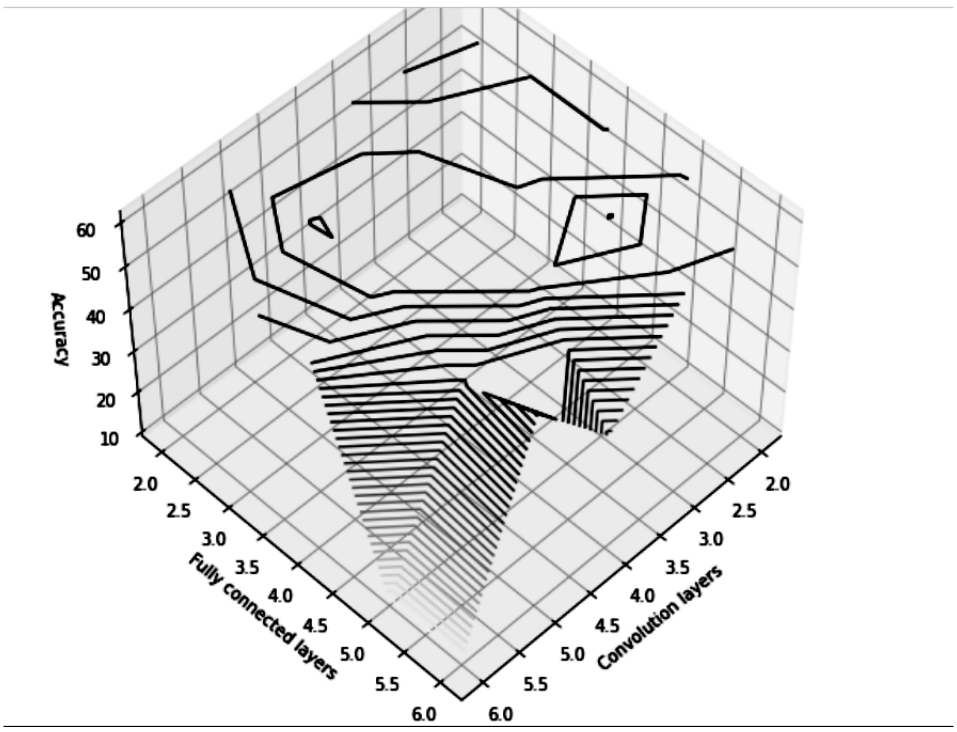


Figure 1: Convolution layers vs Fully connected layers vs Accuracy

We can observer here few things

- When number of fully connected layers increase more than convolution layers, accuracy takes a backseat.
- Accuracy of an architecture with 2 convolution layers and 2 fully connected layers is quite descent.
- Accuracy of architecture with convolution layers = 5 and fully connected layers = 3 is maximum.

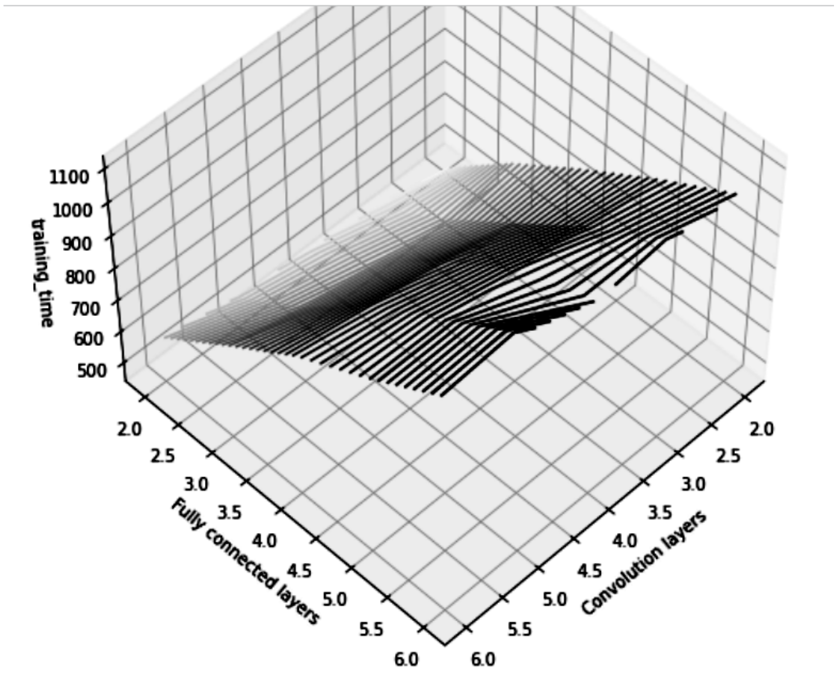


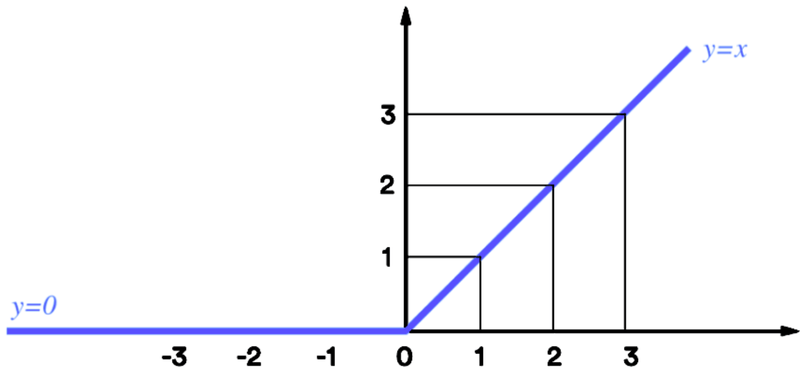
Figure 2: Convolution layers vs Fully connected layers vs Training time (seconds)

We can observe the training time increases with increase in number of layers and complexity of the model.

(b) **Relu vs Sigmoid vs Tanh**

- ReLU

  The rectified linear activation function or ReLU is a piece wise linear function that will output the input directly if it is positive, otherwise, it will output zero.

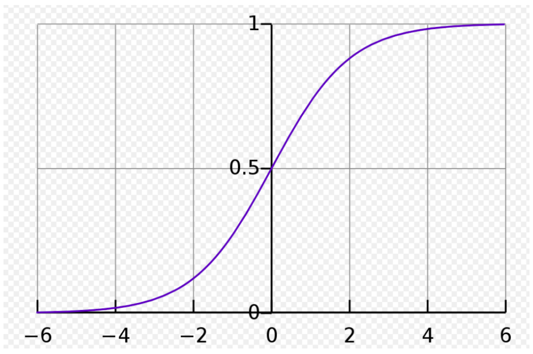


  The derivative of ReLU can be either 1 or 0. Therefore it solves the problem of vanishing and exploding gradients.
  But it can have a problem known as dead activation which is caused by derivative values = 0.

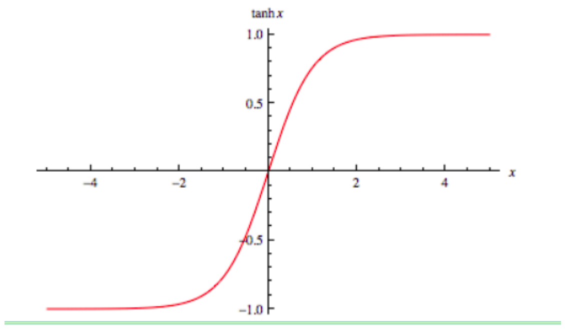
It converges faster to the solution.
Calculation of derivative of ReLU is superfast. It's just an if else condition.
Variations of ReLU are: softplus ReLU, leaky ReLU, noisy ReLU.

- Sigmoid & Tanh

  These are our standard activation functions which were very popular in late 80s-90s.

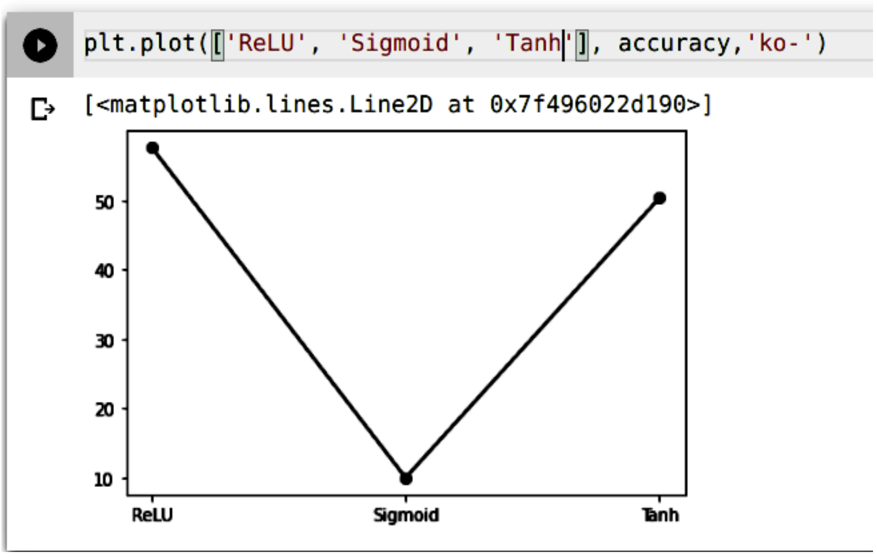


(a) Sigmoid



(b) Tanh

- Result of comparisons



```
plt.plot(['ReLU', 'Sigmoid', 'Tanh'], accuracy,'ko-')
```

```
[<matplotlib.lines.Line2D at 0x7f496022d190>]
```

Figure 4: ReLU vs Sigmoid vs Tanh for accuracy

The results are for same number of iterations. We can observe that sigmoid and tanh will take more time converge at this rate.

(c) SGD with Momentum

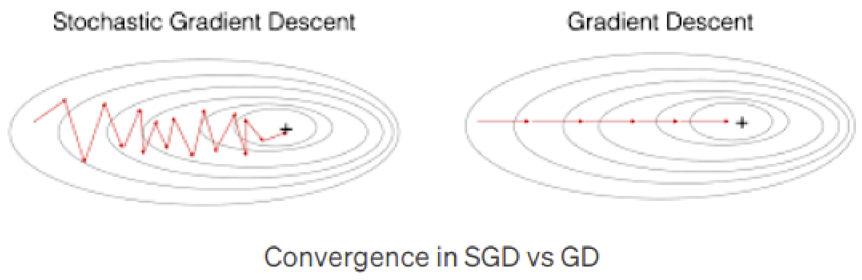


Figure 5: Image Ref : Abhishek M Medium

SGD is by nature noisy as compared to plain GD.

We want some kind of 'moving' average which would 'denoise' the data and bring it closer to the original function.

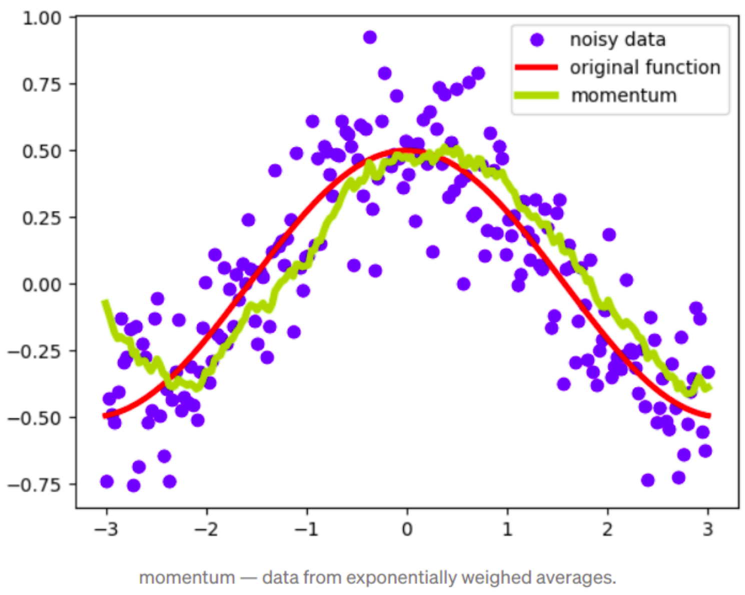
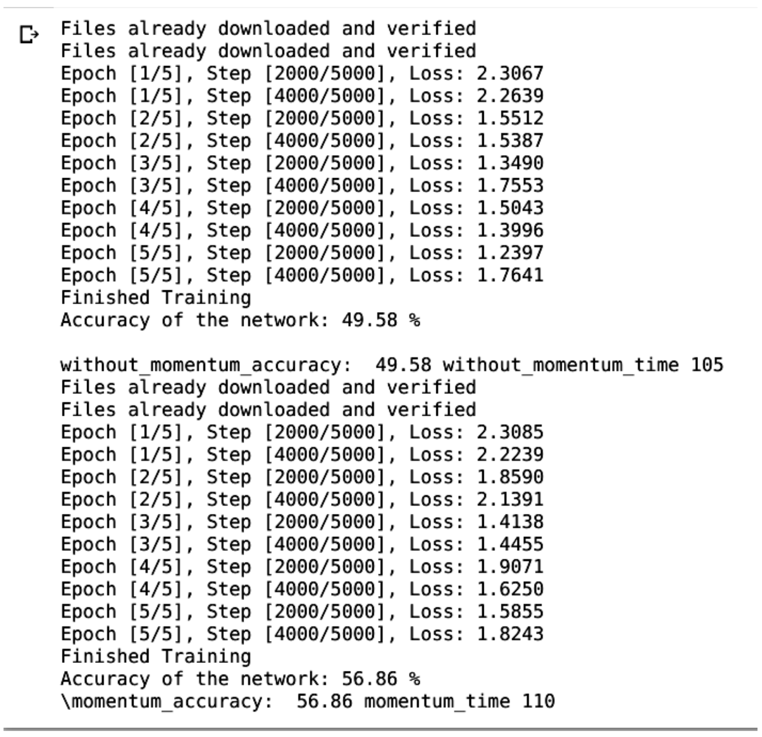


Figure 6: Image Ref : Vitaly B TowardDataScience

So basically SGD plus momentum speeds up out convergence with the concept of moving averages.

Results are shown as below for with and without momentum for same number of iterations.



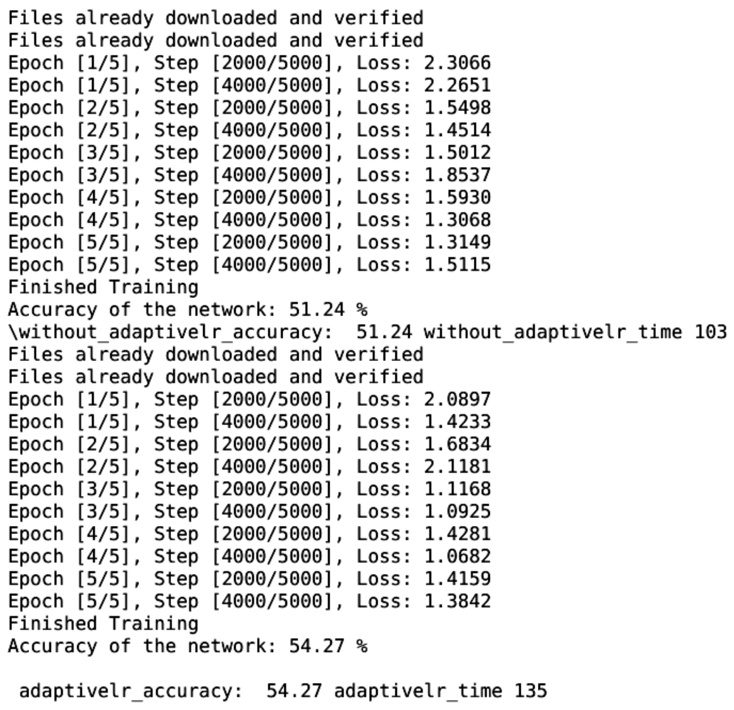
```
Files already downloaded and verified
Files already downloaded and verified
Epoch [1/5], Step [2000/5000], Loss: 2.3067
Epoch [1/5], Step [4000/5000], Loss: 2.2639
Epoch [2/5], Step [2000/5000], Loss: 1.5512
Epoch [2/5], Step [4000/5000], Loss: 1.5387
Epoch [3/5], Step [2000/5000], Loss: 1.3490
Epoch [3/5], Step [4000/5000], Loss: 1.7553
Epoch [4/5], Step [2000/5000], Loss: 1.5043
Epoch [4/5], Step [4000/5000], Loss: 1.3996
Epoch [5/5], Step [2000/5000], Loss: 1.2397
Epoch [5/5], Step [4000/5000], Loss: 1.7641
Finished Training
Accuracy of the network: 49.58 %

without_momentum_accuracy:  49.58 without_momentum_time 105
Files already downloaded and verified
Files already downloaded and verified
Epoch [1/5], Step [2000/5000], Loss: 2.3085
Epoch [1/5], Step [4000/5000], Loss: 2.2239
Epoch [2/5], Step [2000/5000], Loss: 1.8590
Epoch [2/5], Step [4000/5000], Loss: 2.1391
Epoch [3/5], Step [2000/5000], Loss: 1.4138
Epoch [3/5], Step [4000/5000], Loss: 1.4455
Epoch [4/5], Step [2000/5000], Loss: 1.9071
Epoch [4/5], Step [4000/5000], Loss: 1.6250
Epoch [5/5], Step [2000/5000], Loss: 1.5855
Epoch [5/5], Step [4000/5000], Loss: 1.8243
Finished Training
Accuracy of the network: 56.86 %
\momentum_accuracy:  56.86 momentum_time 110
```

Figure 7: with momentum v/s without momentum

We are able to converge faster towards the minima while using momentum. By faster I mean we get greater accuracy for the same number of iterations.

(d) **SGD with Adaptive learning rate**

The performance of the model on the training dataset can be monitored by the learning algorithm and the learning rate can be adjusted in response.
This is called an adaptive learning rate.
Here we use torch.optim.Adadelta to achieve the same.
Results are shown below.

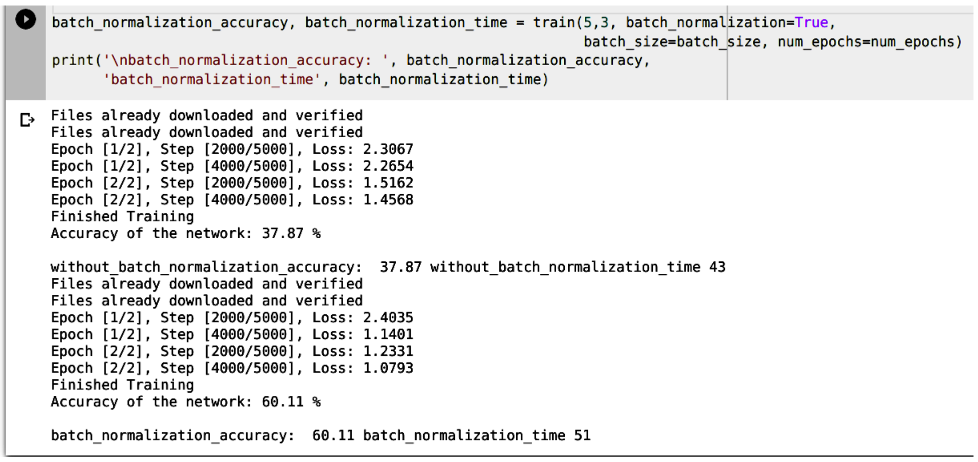


```
Files already downloaded and verified
Files already downloaded and verified
Epoch [1/5], Step [2000/5000], Loss: 2.3066
Epoch [1/5], Step [4000/5000], Loss: 2.2651
Epoch [2/5], Step [2000/5000], Loss: 1.5498
Epoch [2/5], Step [4000/5000], Loss: 1.4514
Epoch [3/5], Step [2000/5000], Loss: 1.5012
Epoch [3/5], Step [4000/5000], Loss: 1.8537
Epoch [4/5], Step [2000/5000], Loss: 1.5930
Epoch [4/5], Step [4000/5000], Loss: 1.3068
Epoch [5/5], Step [2000/5000], Loss: 1.3149
Epoch [5/5], Step [4000/5000], Loss: 1.5115
Finished Training
Accuracy of the network: 51.24 %
\without_adaptivelr_accuracy:  51.24 without_adaptivelr_time 103
Files already downloaded and verified
Files already downloaded and verified
Epoch [1/5], Step [2000/5000], Loss: 2.0897
Epoch [1/5], Step [4000/5000], Loss: 1.4233
Epoch [2/5], Step [2000/5000], Loss: 1.6834
Epoch [2/5], Step [4000/5000], Loss: 2.1181
Epoch [3/5], Step [2000/5000], Loss: 1.1168
Epoch [3/5], Step [4000/5000], Loss: 1.0925
Epoch [4/5], Step [2000/5000], Loss: 1.4281
Epoch [4/5], Step [4000/5000], Loss: 1.0682
Epoch [5/5], Step [2000/5000], Loss: 1.4159
Epoch [5/5], Step [4000/5000], Loss: 1.3842
Finished Training
Accuracy of the network: 54.27 %

 adaptivelr_accuracy:  54.27 adaptivelr_time 135
```

Figure 8: adaptive learning rate vs constant learning rate for accuracy

(e) SGD with Batch Normalization

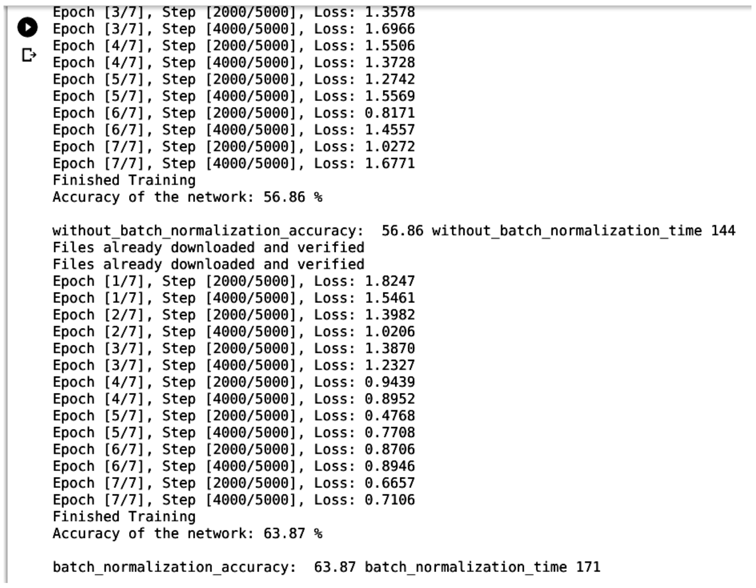
Batch normalization is an idea which was discovered in late 2015s. It is a very interesting idea.
In case of deep networks we can place a normalization layer before our standard layers in order to normalize the income input data.
Batch norm solves the problem of internal co variance shift.
Results are shown below



```
batch_normalization_accuracy, batch_normalization_time = train(5,3, batch_normalization=True,
                                                              batch_size=batch_size, num_epochs=num_epochs)
print('\nbatch_normalization_accuracy: ', batch_normalization_accuracy,
      'batch_normalization_time', batch_normalization_time)
```

```
Files already downloaded and verified
Files already downloaded and verified
Epoch [1/2], Step [2000/5000], Loss: 2.3067
Epoch [1/2], Step [4000/5000], Loss: 2.2654
Epoch [2/2], Step [2000/5000], Loss: 1.5162
Epoch [2/2], Step [4000/5000], Loss: 1.4568
Finished Training
Accuracy of the network: 37.87 %

without_batch_normalization_accuracy:  37.87 without_batch_normalization_time 43
Files already downloaded and verified
Files already downloaded and verified
Epoch [1/2], Step [2000/5000], Loss: 2.4035
Epoch [1/2], Step [4000/5000], Loss: 1.1401
Epoch [2/2], Step [2000/5000], Loss: 1.2331
Epoch [2/2], Step [4000/5000], Loss: 1.0793
Finished Training
Accuracy of the network: 60.11 %

batch_normalization_accuracy:  60.11 batch_normalization_time 51
```

Figure 9: Notice how fast the results are converging for only 2 iterations.



```
Epoch [3/7], Step [2000/5000], Loss: 1.3578
Epoch [3/7], Step [4000/5000], Loss: 1.6966
Epoch [4/7], Step [2000/5000], Loss: 1.5506
Epoch [4/7], Step [4000/5000], Loss: 1.3728
Epoch [5/7], Step [2000/5000], Loss: 1.2742
Epoch [5/7], Step [4000/5000], Loss: 1.5569
Epoch [6/7], Step [2000/5000], Loss: 0.8171
Epoch [6/7], Step [4000/5000], Loss: 1.4557
Epoch [7/7], Step [2000/5000], Loss: 1.0272
Epoch [7/7], Step [4000/5000], Loss: 1.6771
Finished Training
Accuracy of the network: 56.86 %

without_batch_normalization_accuracy:  56.86 without_batch_normalization_time 144
Files already downloaded and verified
Files already downloaded and verified
Epoch [1/7], Step [2000/5000], Loss: 1.8247
Epoch [1/7], Step [4000/5000], Loss: 1.5461
Epoch [2/7], Step [2000/5000], Loss: 1.3982
Epoch [2/7], Step [4000/5000], Loss: 1.0206
Epoch [3/7], Step [2000/5000], Loss: 1.3870
Epoch [3/7], Step [4000/5000], Loss: 1.2327
Epoch [4/7], Step [2000/5000], Loss: 0.9439
Epoch [4/7], Step [4000/5000], Loss: 0.8952
Epoch [5/7], Step [2000/5000], Loss: 0.4768
Epoch [5/7], Step [4000/5000], Loss: 0.7708
Epoch [6/7], Step [2000/5000], Loss: 0.8706
Epoch [6/7], Step [4000/5000], Loss: 0.8946
Epoch [7/7], Step [2000/5000], Loss: 0.6657
Epoch [7/7], Step [4000/5000], Loss: 0.7106
Finished Training
Accuracy of the network: 63.87 %

batch_normalization_accuracy:  63.87 batch_normalization_time 171
```

Figure 10: Notice batch normalization still performs better when iterations are increased

Advantages

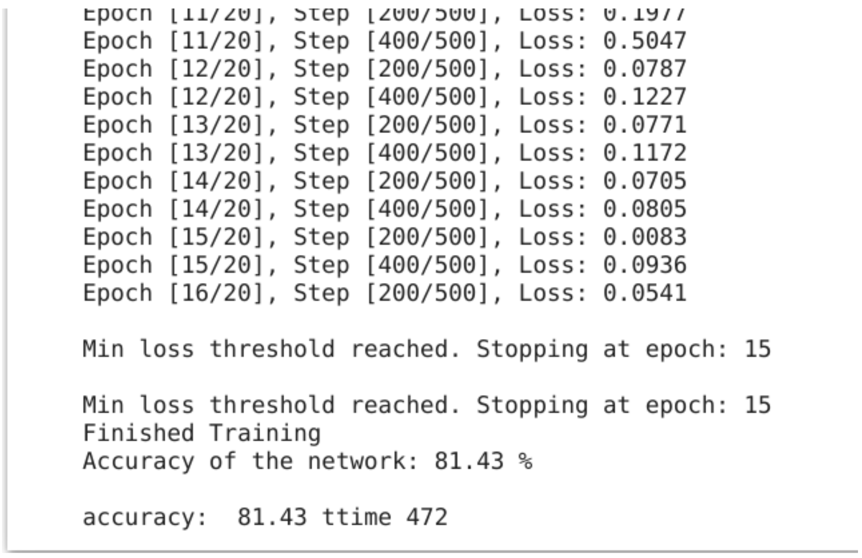
- Faster convergence
- We can afford to have larger learning rates
- Works as a weak regularizer
- Avoids internal co variance shift. Thus allowing us to train deeper NNs.

(f) **Final Architecture**

We will be using an architecture as suggested below in sequential manner on cifar-10 data set. We will use adaptive learning rate, batch normalization, ReLU activation, batch size = 100, max epoch = 20, loss threshold = 0.005

- Input image 3 x 32 x 32
- 2 Convolution layers with kernel size 3 x 3 (number of kernels keep increasing by a factor of 100)
- Max pool layer with kernel size 2 x 2
- 5 Convolution layers with kernel size 3 x 3 (number of kernels keep increasing by a factor of 100 in addition to previous layers)
- Max pool layer with kernel size 2 x 2
- 3 fully connected layers with size 100 increasing with a factor of 100 (This includes the final output layer with only 10 outputs)

Results are as listed below



```
Epoch [11/20], Step [200/500], Loss: 0.1977
Epoch [11/20], Step [400/500], Loss: 0.5047
Epoch [12/20], Step [200/500], Loss: 0.0787
Epoch [12/20], Step [400/500], Loss: 0.1227
Epoch [13/20], Step [200/500], Loss: 0.0771
Epoch [13/20], Step [400/500], Loss: 0.1172
Epoch [14/20], Step [200/500], Loss: 0.0705
Epoch [14/20], Step [400/500], Loss: 0.0805
Epoch [15/20], Step [200/500], Loss: 0.0083
Epoch [15/20], Step [400/500], Loss: 0.0936
Epoch [16/20], Step [200/500], Loss: 0.0541

Min loss threshold reached. Stopping at epoch: 15

Min loss threshold reached. Stopping at epoch: 15
Finished Training
Accuracy of the network: 81.43 %

accuracy:  81.43 ttime 472
```

We were able to achieve an accuracy of **81.43%** on test set. Training time was **7.8 minutes** running on kaggle online GPU.

2. ***Transfer learning (Alexnet as feature extractor)***

Transfer learning is ML technique where a model developed for a first task is then reused for a model on a second task. For example we can train a model to classify birds and cats and then use the same model modified a little bit in the last layers to classify bees and dogs.
This way the first model acts as a feature extractor for the second model.

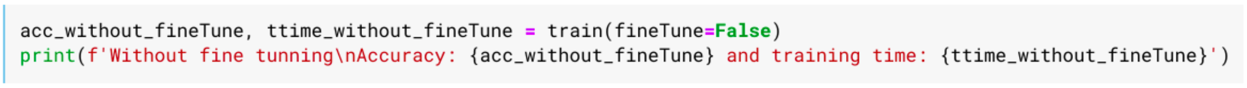
The popularity of this approach comes from the fact that it is very fast to build new models using this as we do not need to again retrain the weights for common features.

For our experiment we will be using the following data set available on Kaggle Hymen optera data : It contains 2 classes of ants and bees.

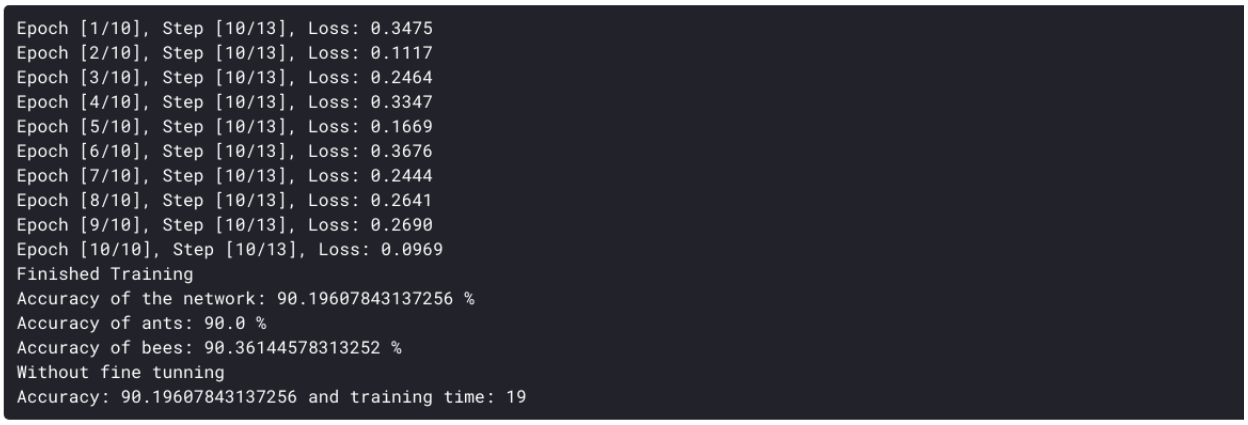
We will be using **Alexnet** as the base model.

(a) **Without fine tuning**

In this approach we do not freeze the base model's parameters
Below are the results obtained on the test set.



```
acc_without_fineTune, ttime_without_fineTune = train(fineTune=False)
print(f'Without fine tunning\nAccuracy: {acc_without_fineTune} and training time: {ttime_without_fineTune}')
```

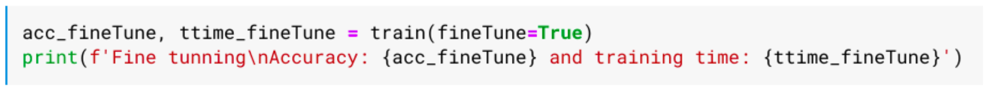


```
Epoch [1/10], Step [10/13], Loss: 0.3475
Epoch [2/10], Step [10/13], Loss: 0.1117
Epoch [3/10], Step [10/13], Loss: 0.2464
Epoch [4/10], Step [10/13], Loss: 0.3347
Epoch [5/10], Step [10/13], Loss: 0.1669
Epoch [6/10], Step [10/13], Loss: 0.3676
Epoch [7/10], Step [10/13], Loss: 0.2444
Epoch [8/10], Step [10/13], Loss: 0.2641
Epoch [9/10], Step [10/13], Loss: 0.2690
Epoch [10/10], Step [10/13], Loss: 0.0969
Finished Training
Accuracy of the network: 90.19607843137256 %
Accuracy of ants: 90.0 %
Accuracy of bees: 90.36144578313252 %
Without fine tunning
Accuracy: 90.19607843137256 and training time: 19
```

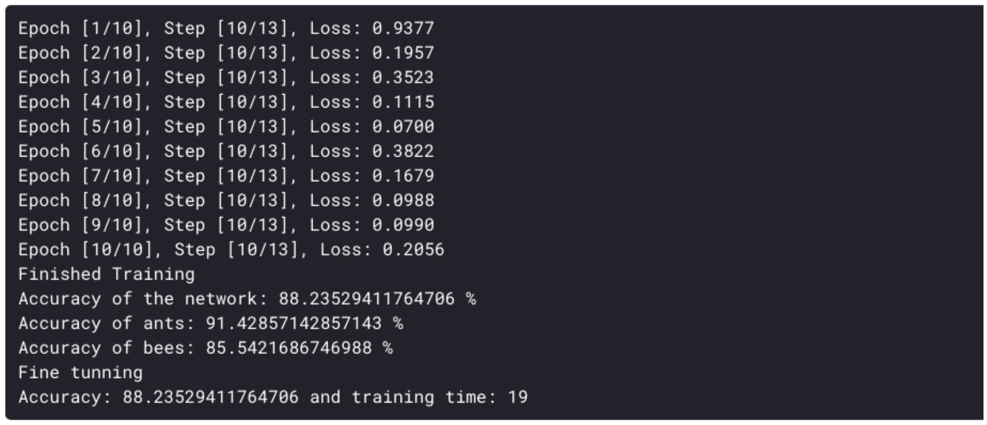
We were able to achieve an accuracy of **90%** on the test set

(b) **With fine tuning**

In this approach we freeze the base model's parameters and only update the weights of the additional layers
Below are the results obtained on the test set.



```
acc_fineTune, ttime_fineTune = train(fineTune=True)
print(f'Fine tunning\nAccuracy: {acc_fineTune} and training time: {ttime_fineTune}')
```



```
Epoch [1/10], Step [10/13], Loss: 0.9377
Epoch [2/10], Step [10/13], Loss: 0.1957
Epoch [3/10], Step [10/13], Loss: 0.3523
Epoch [4/10], Step [10/13], Loss: 0.1115
Epoch [5/10], Step [10/13], Loss: 0.0700
Epoch [6/10], Step [10/13], Loss: 0.3822
Epoch [7/10], Step [10/13], Loss: 0.1679
Epoch [8/10], Step [10/13], Loss: 0.0988
Epoch [9/10], Step [10/13], Loss: 0.0990
Epoch [10/10], Step [10/13], Loss: 0.2056
Finished Training
Accuracy of the network: 88.23529411764706 %
Accuracy of ants: 91.42857142857143 %
Accuracy of bees: 85.5421686746988 %
Fine tunning
Accuracy: 88.23529411764706 and training time: 19
```

We were able to achieve an accuracy of **88%** on the test set