VR PROJECT REPORT

Team:
Ankrutee Arora - IMT2020034
Ashish Gatreddi- IMT2020073
Tejdeep Gutta- IMT2020102

3a. **Results with CNN's on CIFAR-10**

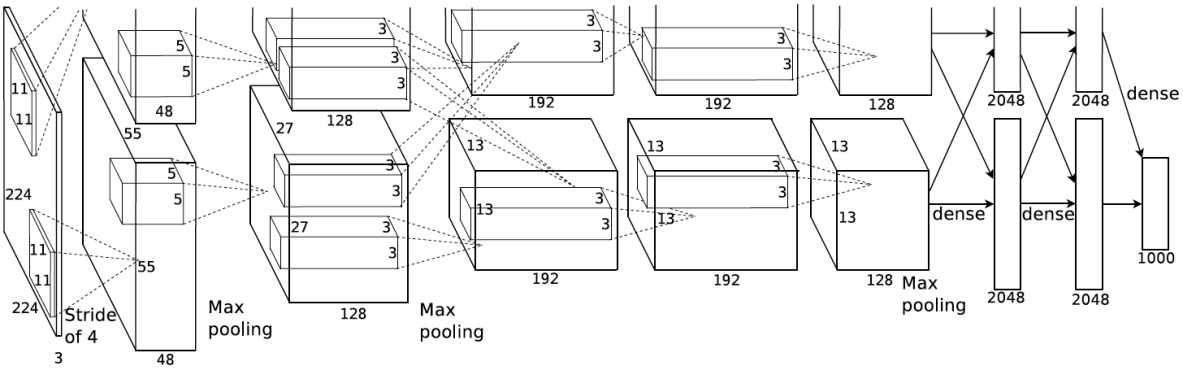
We have previously worked on classification on the CIFAR-10 dataset using models like SVM and some classical neural networks. Classical neural networks although can be used for tasks like image classification, the issue is that they work fine only for small sized images and become extremely inefficient when applied to medium or large images. The reason for that is the huge number of parameters required by classical neural networks. This problem is dealt with by using Convolutional neural networks since they implement partially connected layers and weight sharing.

The main components of a CNN are :

1. Convolutional layers
2. Pooling layers
3. Parameter sharing

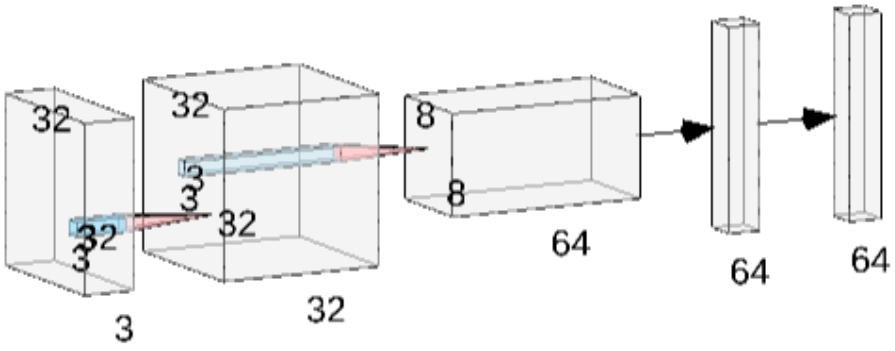
We also have fully connected layers at the end, the fully connected layer helps to map the representation between the input and the output.

One of the best known Convolutional Neural Networks is AlexNet, it was used for image classification on ImageNet dataset. The architecture used in AlexNet is given below. It used two GPU's and had a training time of 5-6 days.



AlexNet has 5 convolutional layers and 3 fully connected layers and it uses the ReLU activation function.
We start with a simple medium depth CNN which consists of 2 convolutional and 2 fully connected layers, we then slowly increase the depth and also vary the optimiser between ADAM and SGD.

First version :



The architecture is given as follows :

(0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

(1): ReLU()

(2): MaxPool2d(kernel_size=4, stride=4, padding=0, dilation=1, ceil_mode=False)

(3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

(4): ReLU()

(5): MaxPool2d(kernel_size=8, stride=8, padding=0, dilation=1, ceil_mode=False)

(6): Flatten(start_dim=1, end_dim=-1)

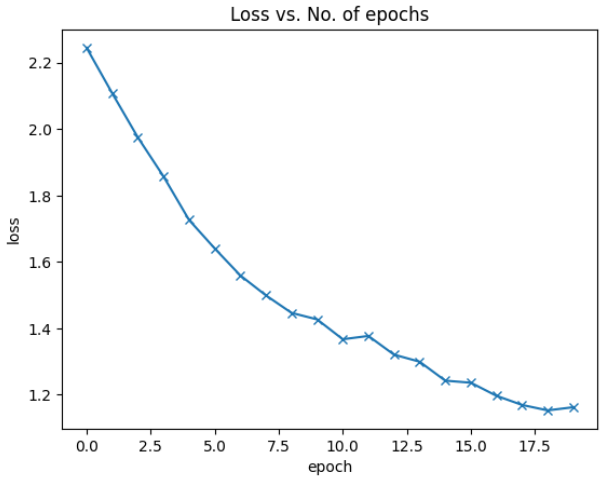
(7): Linear(in_features=64, out_features=64, bias=True)

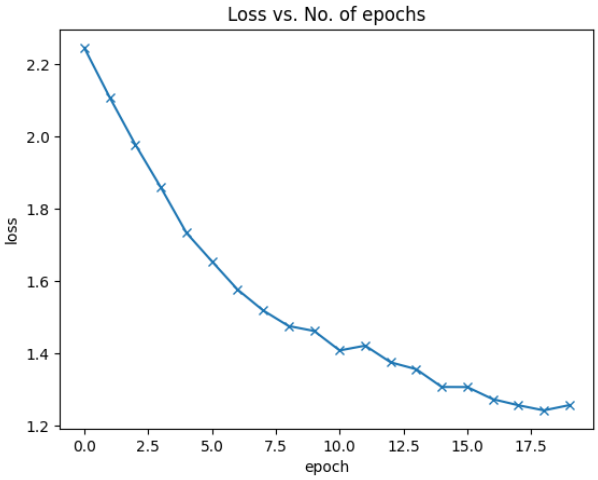
(8): ReLU()

(9): Linear(in_features=64, out_features=10, bias=True)

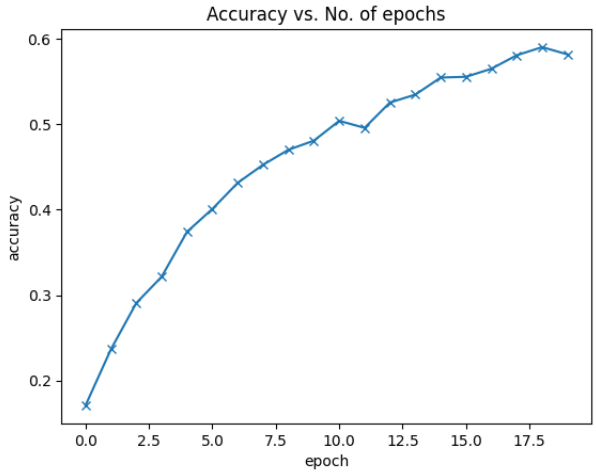
Using this architecture with a batch size of 1000, and training over 20 epochs, we obtain the following results,

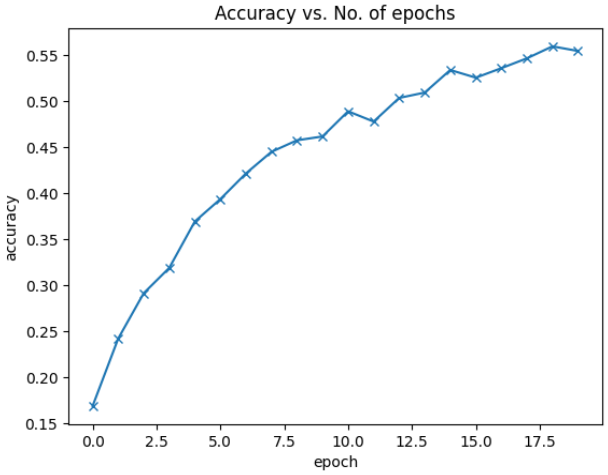
here we have used SGD as the optimiser with learning rate 0.01 and momentum 0.9





Left side - loss due to train data right side - loss due to validation data

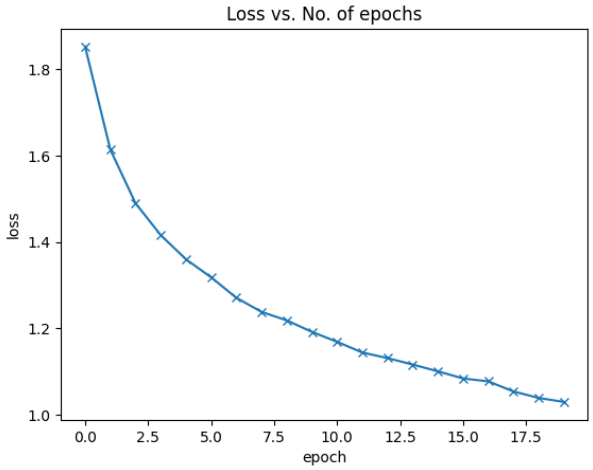


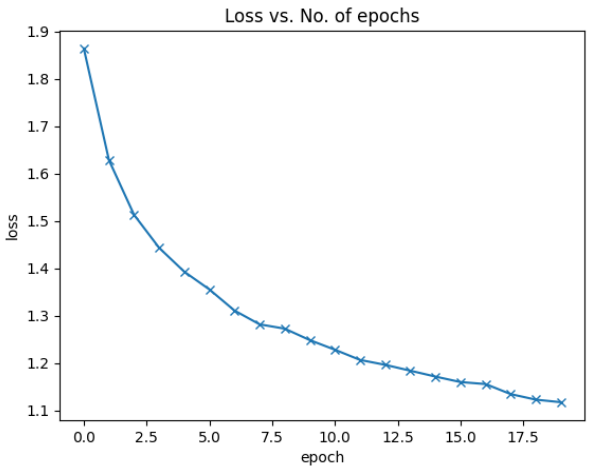


Left side - Accuracy due to train data right side - Accuracy due to validation data

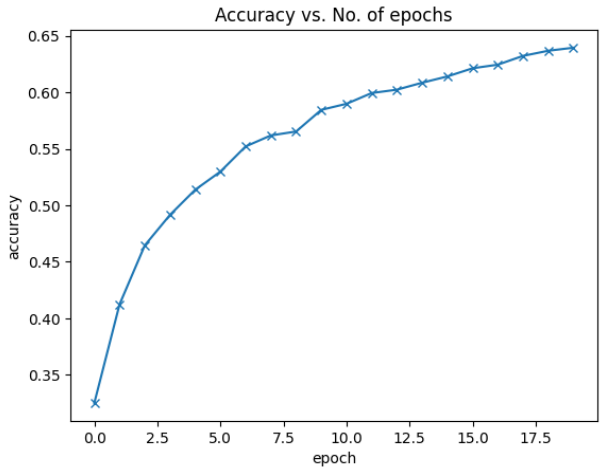
Upon testing we get an accuracy of **55.55%.**

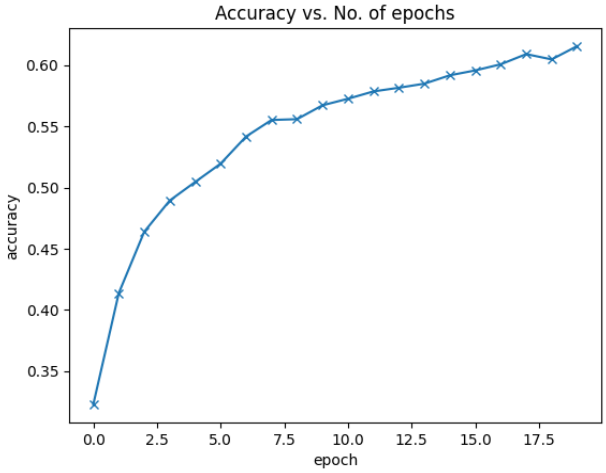
Now, we use the same architecture but this time the optimiser used is ADAM and the parameters are the default ones given by torch.optim and learning rate 0.0001





Left side - loss due to train data right side - loss due to validation data

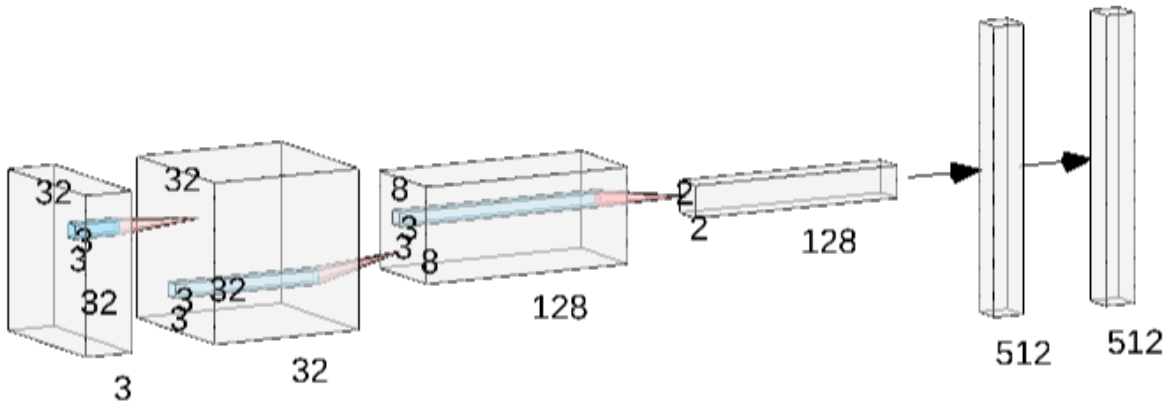




Left side - Accuracy due to train data right side - Accuracy due to validation data

Upon testing we get an accuracy of **62.72%**

Second version : (adam done)



The architecture is given as follows :

(0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

(1): ReLU()

(2): MaxPool2d(kernel_size=4, stride=4, padding=0, dilation=1, ceil_mode=False)

(3): Conv2d(32, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

(4): ReLU()

(5): MaxPool2d(kernel_size=4, stride=4, padding=0, dilation=1, ceil_mode=False)

(6): Flatten(start_dim=1, end_dim=-1)

(7): Linear(in_features=512, out_features=512, bias=True)

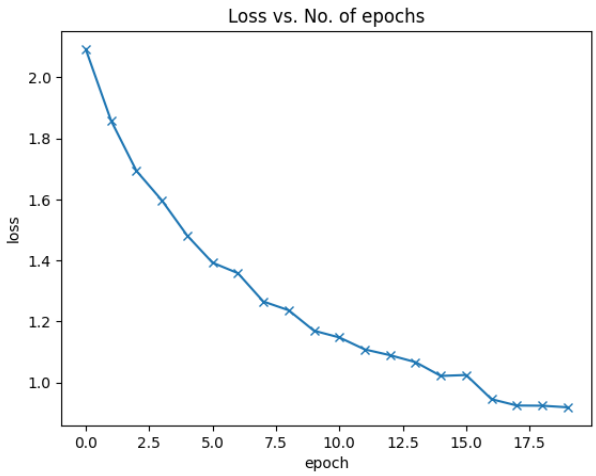
(8): ReLU()

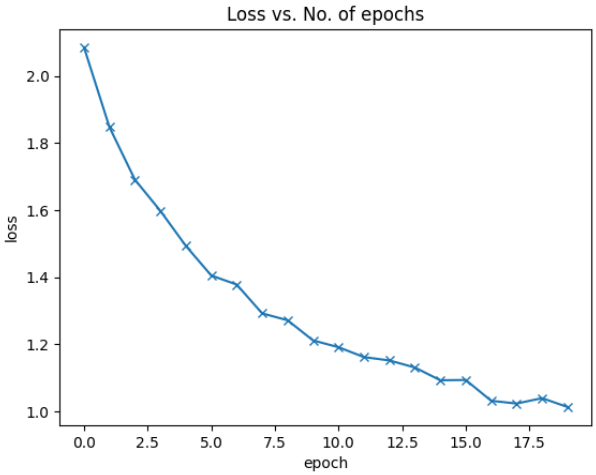
(9): Linear(in_features=512, out_features=10, bias=True)

This time we have kept the number of convolutional and fully connected layers the same but done max pooling in such a way that we have more features after flattening the last convolutional layer, i.e, we have reduced the area of max pooling filter.

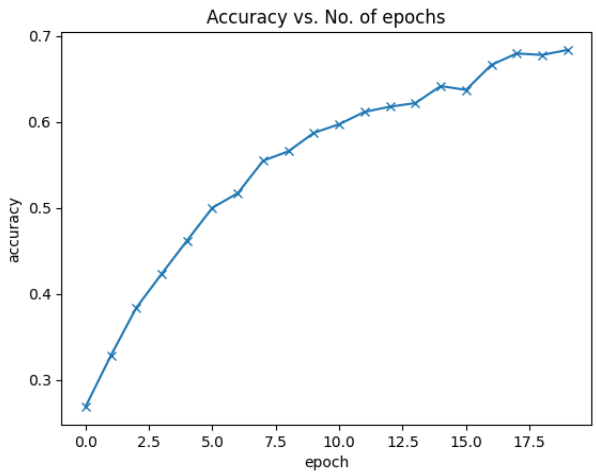
Using this architecture with a batch size of 1000, and training over 20 epochs, we obtain the following result.

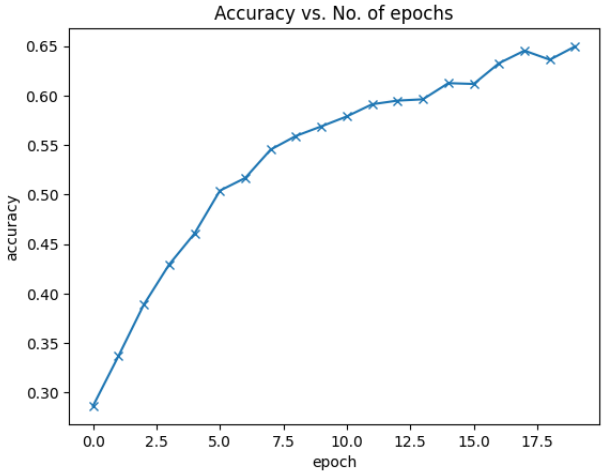
Here we have used SGD as the optimiser with learning rate 0.001 and momentum 0.9





Left side - loss due to train data right side - loss due to validation data

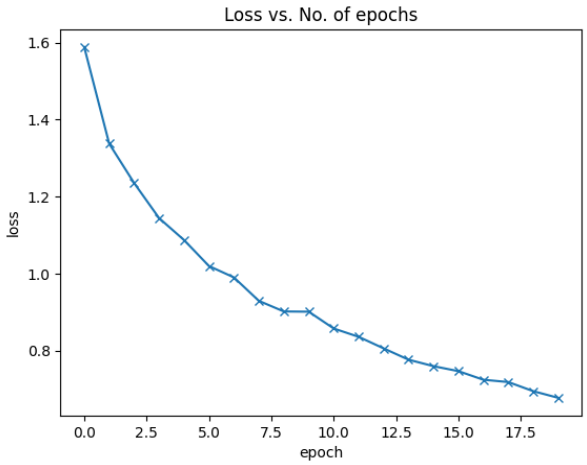


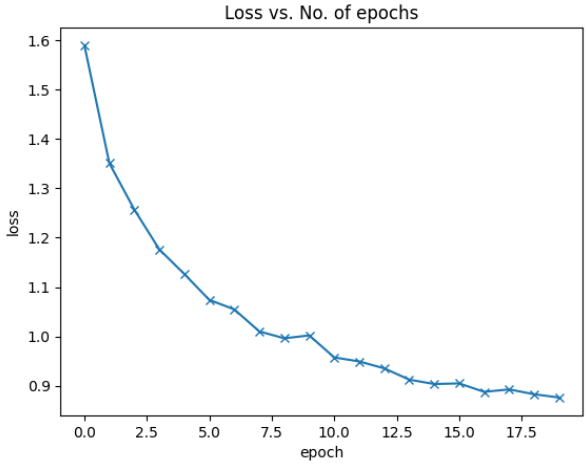


Left side - Accuracy due to train data right side - Accuracy due to validation data

Upon testing we get an accuracy of **63.93%.**

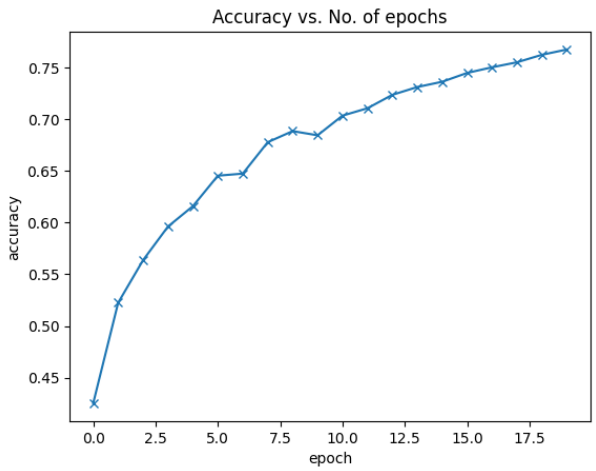
Now, we use the same architecture but this time the optimiser used is ADAM and the parameters are the default ones given by torch.optim with learning rate 0.0001

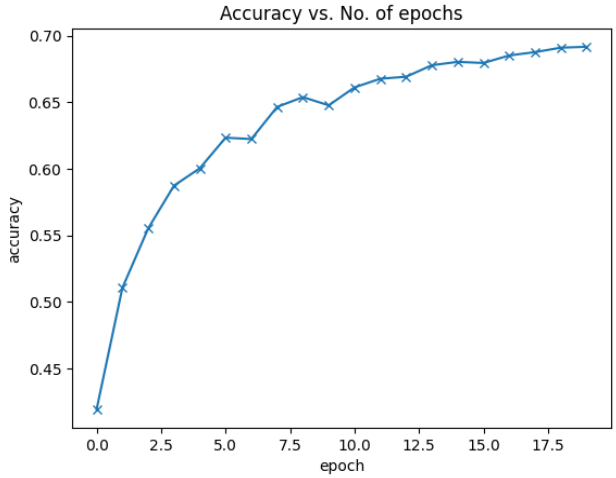




Left side - loss due to train data

right side - loss due to validation data



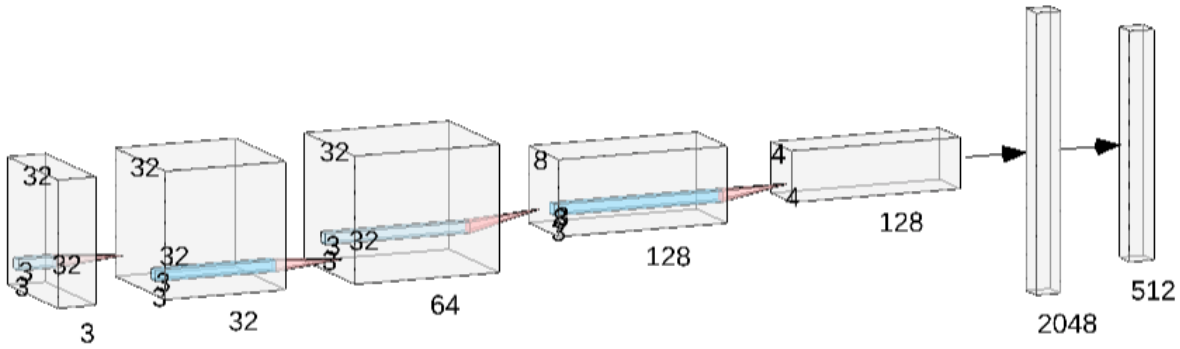


Left side - Accuracy due to train data

right side - Accuracy due to validation data

Upon testing we get an accuracy of **70.48%**

Third version (Best testing accuracy so far):



The architecture is given as follows :

(0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

(1): ReLU()

(2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

(3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

(4): ReLU()

(5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

(6): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

(7): ReLU()

(8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

(9): Flatten(start_dim=1, end_dim=-1)

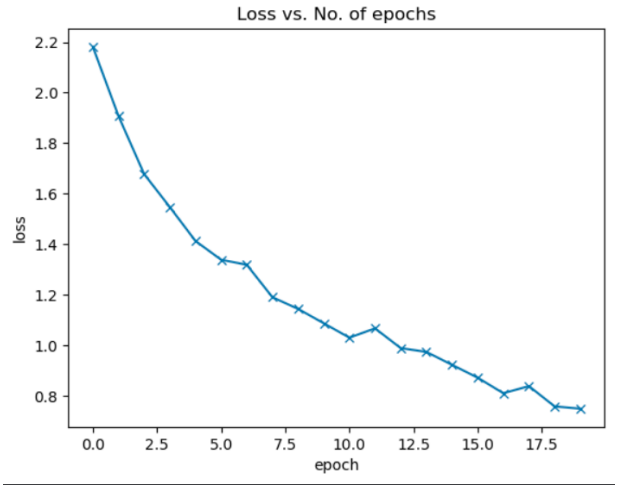
(10): Linear(in_features=2048, out_features=512, bias=True)

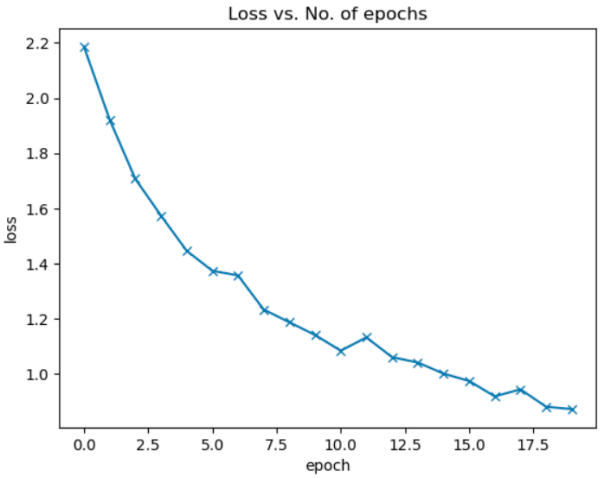
(11): ReLU()

(12): Linear(in_features=512, out_features=10, bias=True)

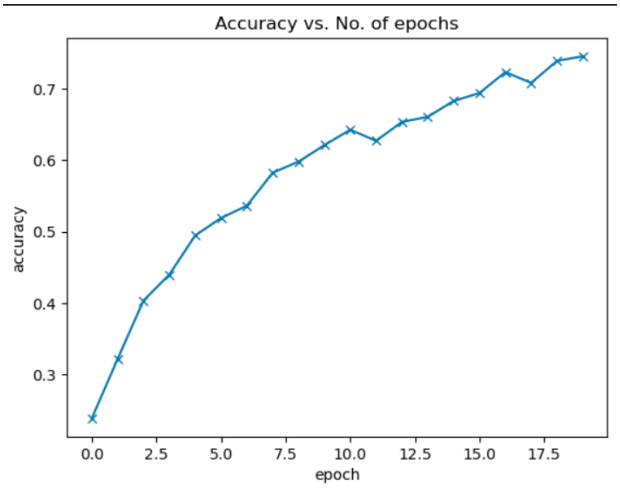
We have introduced a new convolutional layer and also done pooling in multiple steps. Using this architecture with a batch size of 1000, and training over 20 epochs, we obtain

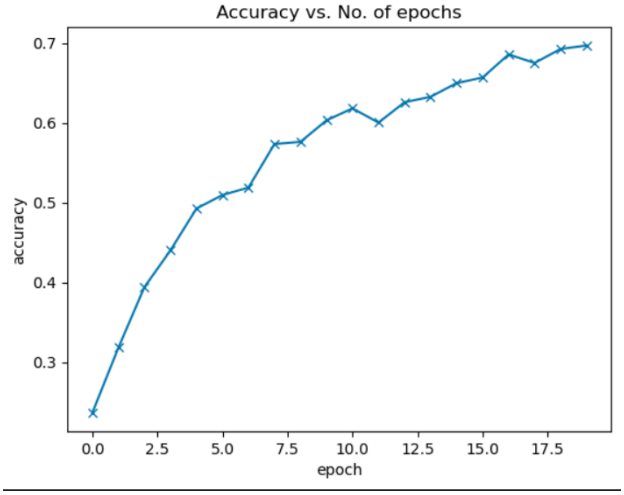
the following results, here we have used SGD as the optimiser with learning rate 0.01 and momentum 0.9 -





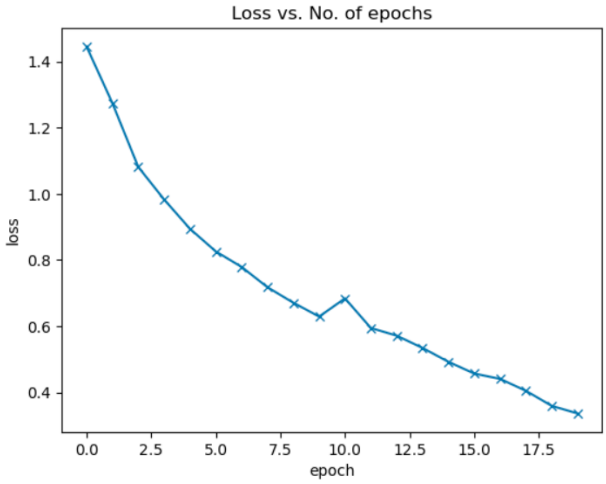
Where the x axis in both graphs represents the number of epochs and the y axis represents the losses of training and validation sets respectively.
We also have the accuracies (training and validation left to right) -

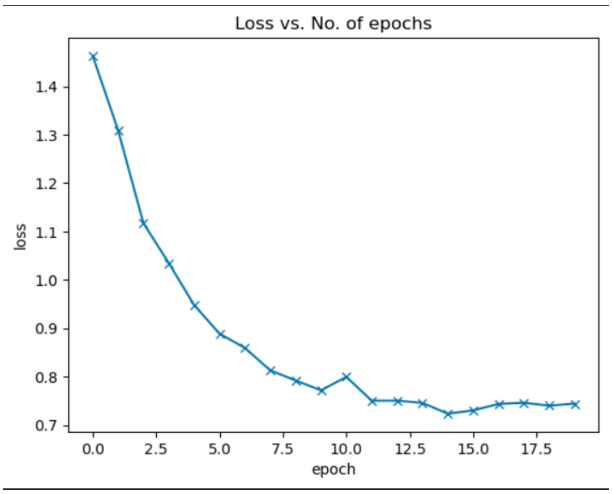




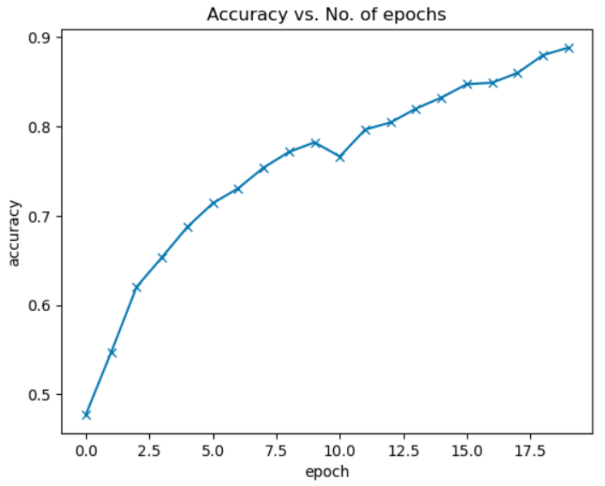
Upon testing we get an accuracy of **69.34%.**

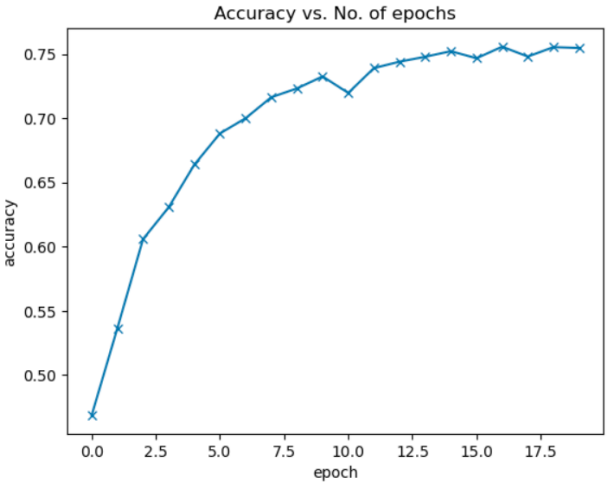
Now, we use the same architecture but this time the optimiser used is ADAM and the learning rate is 0.001-





Above are the loss graphs of training and validation sets respectively (left to right).
We also have the accuracy graphs below -





Upon testing we get an accuracy of **76.39%.**

Here we can observe the convergence to optimal solution is faster in the third model when compared to the first and second models and with increase in epochs further for first and second models there is an approx 5 percent increase in the accuracy.

In all of the above architectures, we have used ReLU as the non-linear activation function, given below is a comparison of ReLU, tanh and sigmoid. With Adam optimizer.

	ReLU accuracy	Tanh accuracy	Sigmoid accuracy
First version	62.72	62.4	43.5
Second version	70.48	70.02	50.11
Third version	76.39	71.75	59.93

We see that Sigmoid performs the worst out of the three for the same parameters, in general the major disadvantage with using Sigmoid is the problem of **vanishing gradients,** this also occurs with tanh**.** It also has slow convergence, and the optimisation process is harder because its outputs are not zero centered.

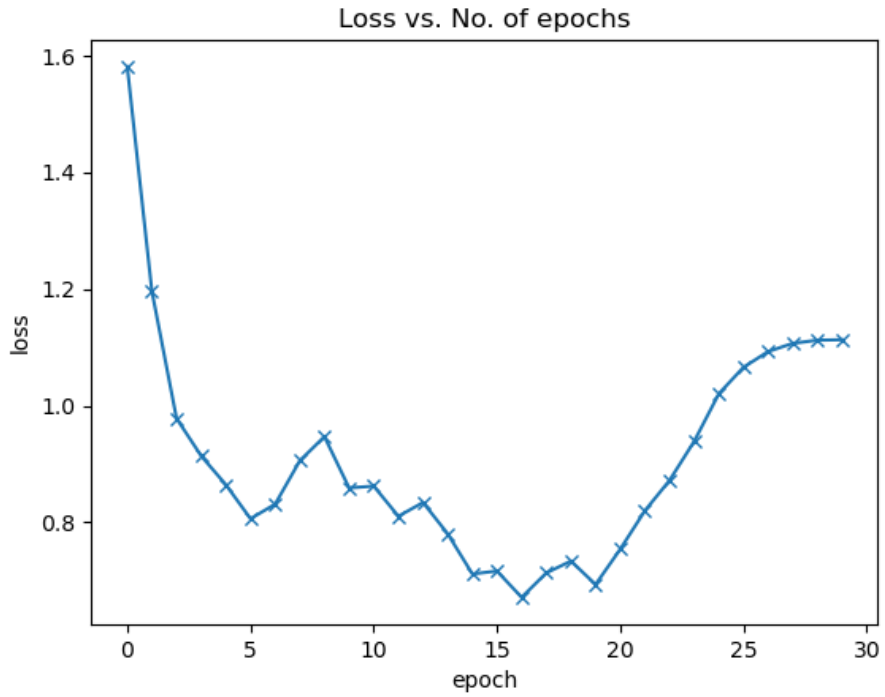
Tanh and ReLU have similar accuracies with ReLU being the better performer out of the two. ReLU is a great alternative to both tanh and sigmoid activation functions. It also converges faster than both tanh and sigmoid. The ReLU function has a fixed derivative (slope) for one linear component and a zero derivative for the other linear component. Therefore, the learning process is much faster with the ReLU function.

The new models which we tried are:

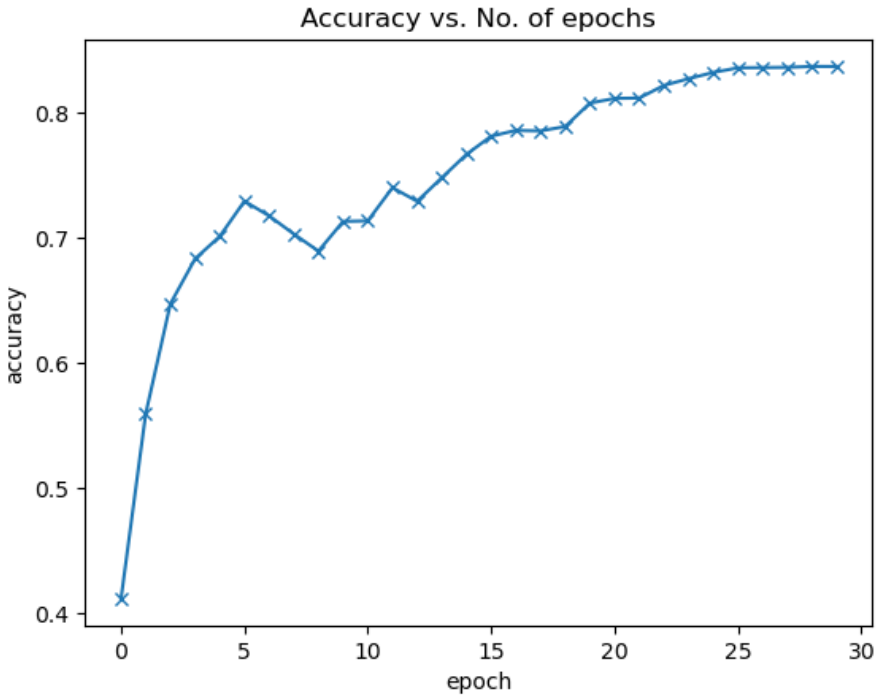
```
----------------------------------------------------------------
      Layer (type)               Output Shape         Param #
================================================================
          Conv2d-1           [-1, 32, 32, 32]             896
            ReLU-2           [-1, 32, 32, 32]               0
          Conv2d-3           [-1, 64, 32, 32]          18,496
            ReLU-4           [-1, 64, 32, 32]               0
         MaxPool2d-5           [-1, 64, 16, 16]               0
          Conv2d-6           [-1, 128, 16, 16]          73,856
            ReLU-7           [-1, 128, 16, 16]               0
          Conv2d-8           [-1, 128, 16, 16]         147,584
            ReLU-9           [-1, 128, 16, 16]               0
        MaxPool2d-10            [-1, 128, 8, 8]               0
         Conv2d-11            [-1, 256, 8, 8]         295,168
           ReLU-12            [-1, 256, 8, 8]               0
         Conv2d-13            [-1, 256, 8, 8]         590,080
           ReLU-14            [-1, 256, 8, 8]               0
        MaxPool2d-15            [-1, 256, 4, 4]               0
         Flatten-16                 [-1, 4096]               0
          Linear-17                 [-1, 1024]       4,195,328
            ReLU-18                 [-1, 1024]               0
          Linear-19                  [-1, 512]         524,800
            ReLU-20                  [-1, 512]               0
          Linear-21                   [-1, 10]           5,130
================================================================
Total params: 5,851,338
Trainable params: 5,851,338
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 3.27
Params size (MB): 22.32
Estimated Total Size (MB): 25.61
----------------------------------------------------------------
```

By using ReLU as activation function, 30 epochs and SGD as optimizer

The testing data loss with respect to number of epochs:



The testing data accuracy with respect to number of epochs:

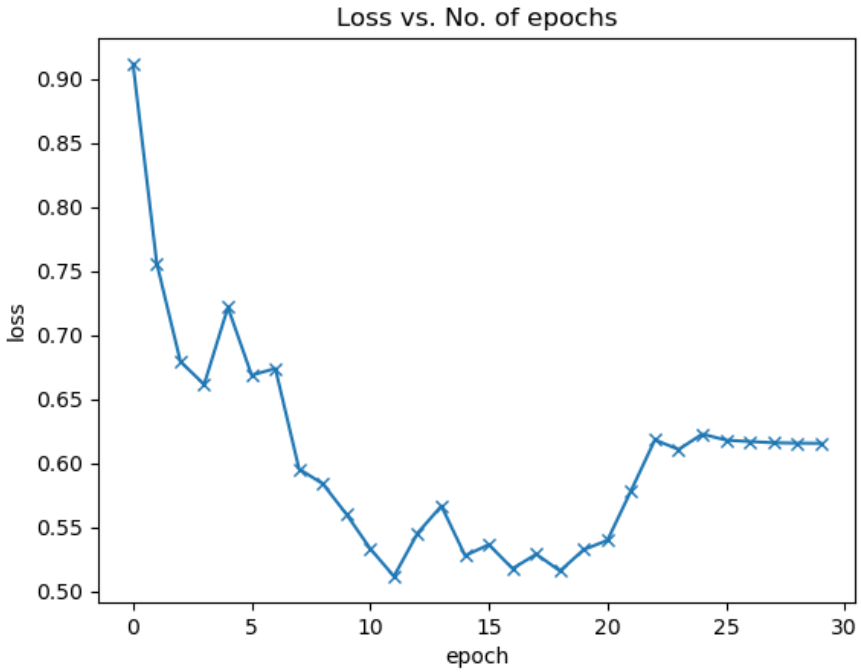


The testing accuracy observed at epoch-15 is approximately 80

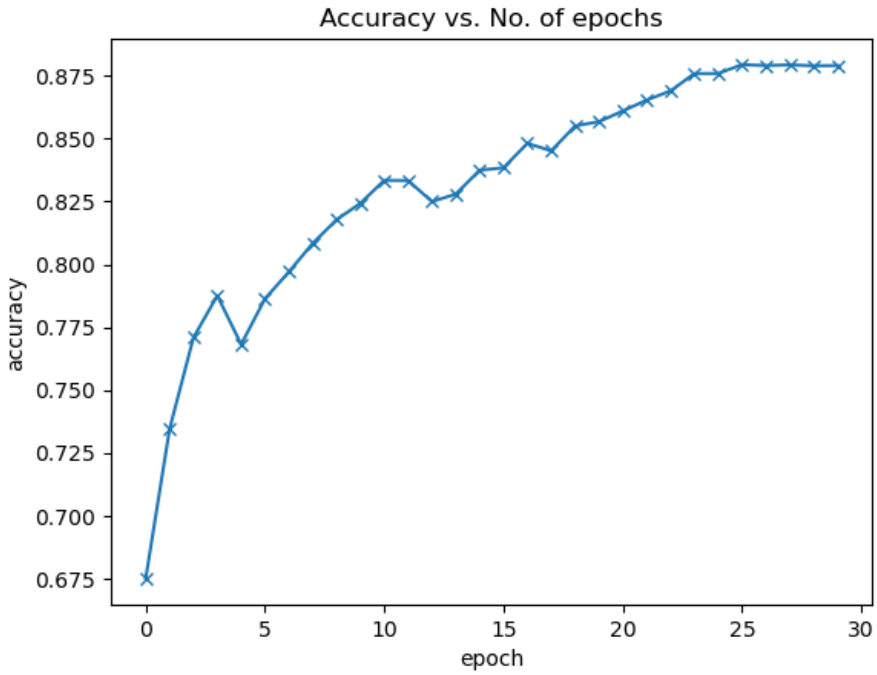
```
----------------------------------------------------------------
      Layer (type)               Output Shape         Param #
================================================================
          Conv2d-1          [-1, 32, 32, 32]             896
            ReLU-2          [-1, 32, 32, 32]               0
          Conv2d-3          [-1, 64, 32, 32]          18,496
            ReLU-4          [-1, 64, 32, 32]               0
       MaxPool2d-5          [-1, 64, 16, 16]               0
     BatchNorm2d-6          [-1, 64, 16, 16]             128
          Conv2d-7         [-1, 128, 16, 16]          73,856
            ReLU-8         [-1, 128, 16, 16]               0
          Conv2d-9         [-1, 128, 16, 16]         147,584
           ReLU-10         [-1, 128, 16, 16]               0
      MaxPool2d-11           [-1, 128, 8, 8]               0
    BatchNorm2d-12           [-1, 128, 8, 8]             256
         Conv2d-13           [-1, 256, 8, 8]         295,168
           ReLU-14           [-1, 256, 8, 8]               0
         Conv2d-15           [-1, 256, 8, 8]         590,080
           ReLU-16           [-1, 256, 8, 8]               0
      MaxPool2d-17           [-1, 256, 4, 4]               0
    BatchNorm2d-18           [-1, 256, 4, 4]             512
        Flatten-19                 [-1, 4096]               0
         Linear-20                 [-1, 1024]       4,195,328
           ReLU-21                 [-1, 1024]               0
         Linear-22                  [-1, 512]         524,800
           ReLU-23                  [-1, 512]               0
         Linear-24                   [-1, 10]           5,130
================================================================
Total params: 5,852,234
Trainable params: 5,852,234
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 3.49
Params size (MB): 22.32
Estimated Total Size (MB): 25.83
----------------------------------------------------------------
```

By using ReLU as activation function, 30 epochs and SGD as optimizer

The testing data loss with respect to epochs:



The testing Accuracy with respect to number of epochs:

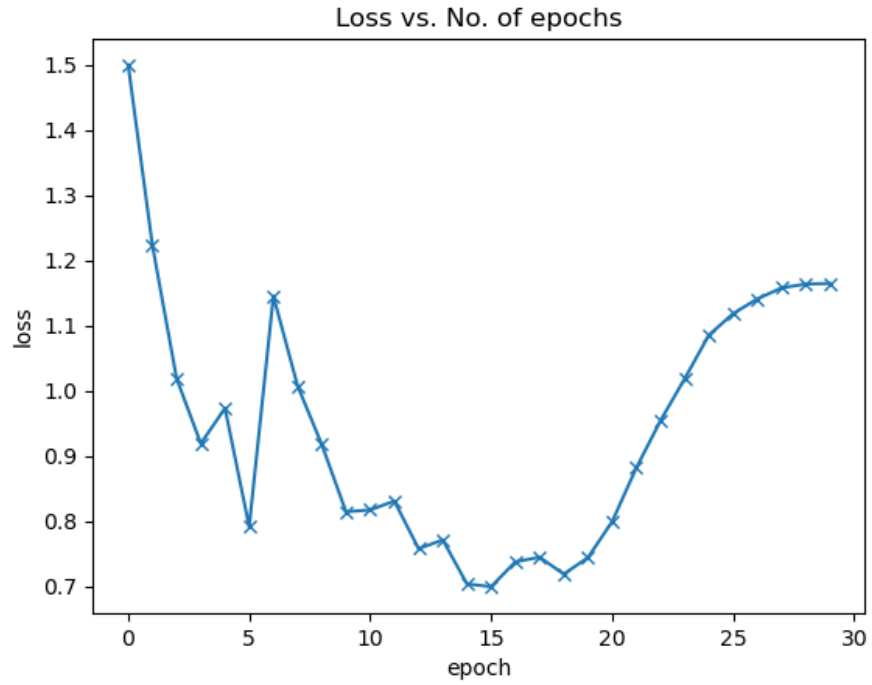


Here we can clearly observe that after 11’th epoch The loss started to increase. testing accuracy received for 11 epochs is: approximately 85 percent.

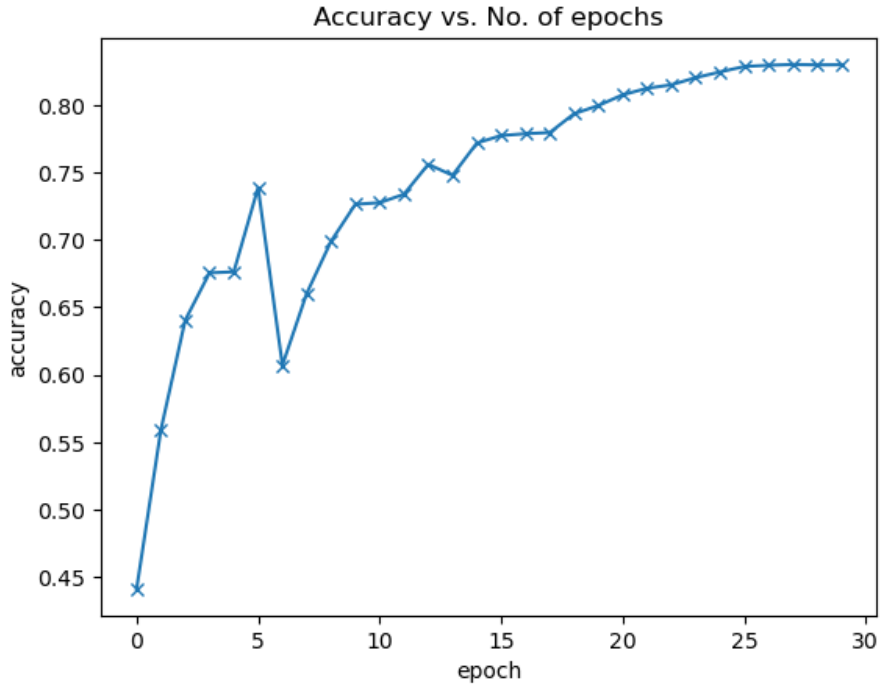
```
----------------------------------------------------------------
      Layer (type)               Output Shape         Param #
================================================================
          Conv2d-1          [-1, 32, 32, 32]             896
            ReLU-2          [-1, 32, 32, 32]               0
          Conv2d-3          [-1, 64, 32, 32]          18,496
            ReLU-4          [-1, 64, 32, 32]               0
       MaxPool2d-5          [-1, 64, 16, 16]               0
          Conv2d-6         [-1, 128, 16, 16]          73,856
            ReLU-7         [-1, 128, 16, 16]               0
          Conv2d-8         [-1, 128, 16, 16]         147,584
            ReLU-9         [-1, 128, 16, 16]               0
      MaxPool2d-10           [-1, 128, 8, 8]               0
         Conv2d-11           [-1, 256, 8, 8]         295,168
           ReLU-12           [-1, 256, 8, 8]               0
         Conv2d-13           [-1, 256, 8, 8]         590,080
           ReLU-14           [-1, 256, 8, 8]               0
      MaxPool2d-15           [-1, 256, 4, 4]               0
        Flatten-16                 [-1, 4096]               0
         Linear-17                 [-1, 1024]       4,195,328
           ReLU-18                 [-1, 1024]               0
         Linear-19                  [-1, 512]         524,800
           ReLU-20                  [-1, 512]               0
         Linear-21                   [-1, 10]           5,130
================================================================
Total params: 5,851,338
Trainable params: 5,851,338
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 3.27
Params size (MB): 22.32
Estimated Total Size (MB): 25.61
----------------------------------------------------------------
```

By using ReLU as activation function, 30 epochs and ADAM as optimizer

The testing data loss with respect to number of epochs:



The testing data accuracy with respect to number of epochs:

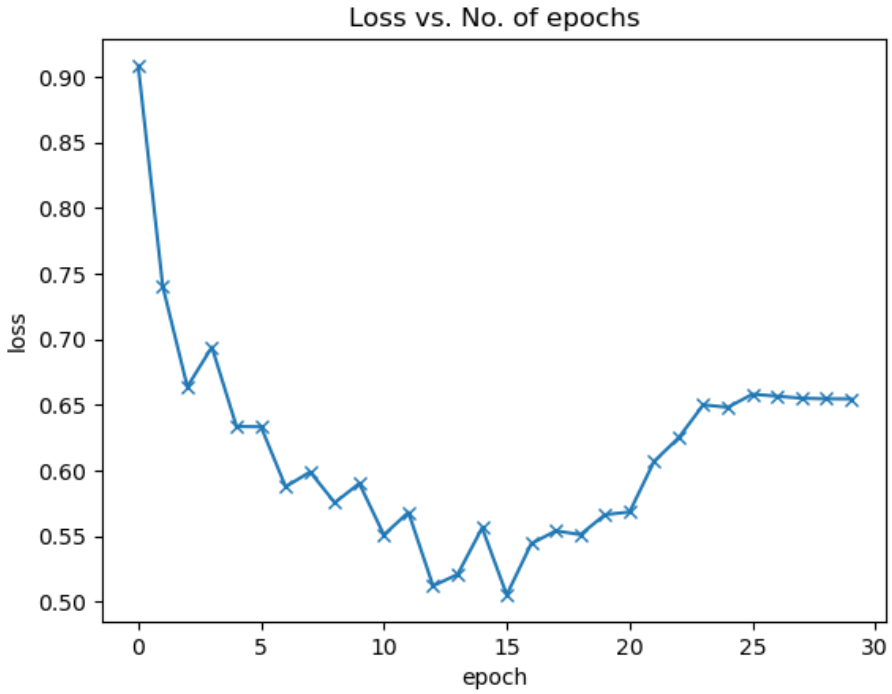


Here we can clearly observe that after 15'th epoch The loss started to increase. testing accuracy received for 15 epochs is: approximately 78 percent.

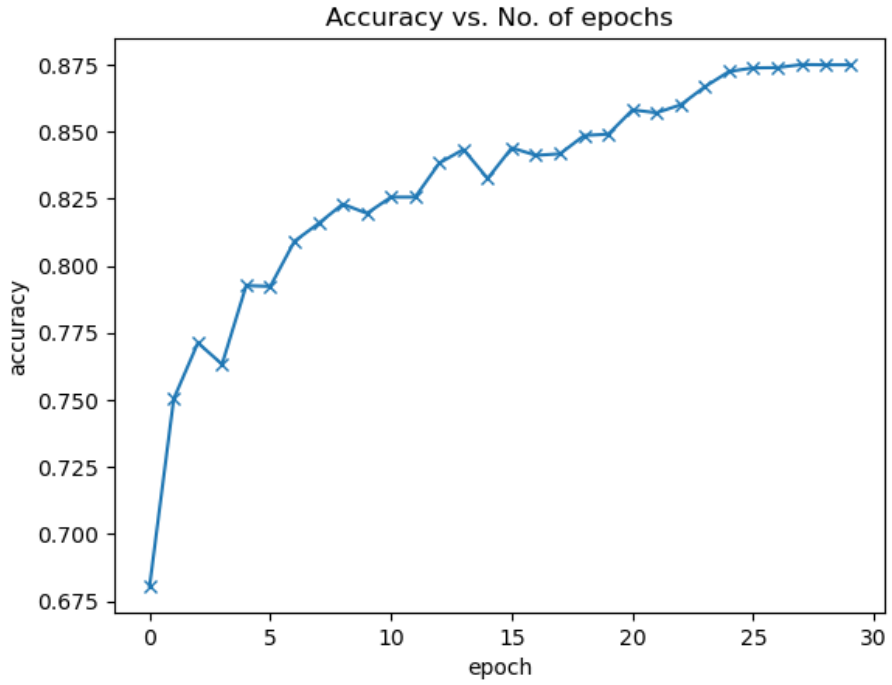
```
----------------------------------------------------------------
      Layer (type)               Output Shape         Param #
================================================================
          Conv2d-1          [-1, 32, 32, 32]             896
            ReLU-2          [-1, 32, 32, 32]               0
          Conv2d-3          [-1, 64, 32, 32]          18,496
            ReLU-4          [-1, 64, 32, 32]               0
       MaxPool2d-5          [-1, 64, 16, 16]               0
     BatchNorm2d-6          [-1, 64, 16, 16]             128
          Conv2d-7         [-1, 128, 16, 16]          73,856
            ReLU-8         [-1, 128, 16, 16]               0
          Conv2d-9         [-1, 128, 16, 16]         147,584
           ReLU-10         [-1, 128, 16, 16]               0
      MaxPool2d-11          [-1, 128, 8, 8]               0
    BatchNorm2d-12          [-1, 128, 8, 8]             256
         Conv2d-13          [-1, 256, 8, 8]         295,168
           ReLU-14          [-1, 256, 8, 8]               0
         Conv2d-15          [-1, 256, 8, 8]         590,080
           ReLU-16          [-1, 256, 8, 8]               0
      MaxPool2d-17          [-1, 256, 4, 4]               0
    BatchNorm2d-18          [-1, 256, 4, 4]             512
        Flatten-19              [-1, 4096]               0
         Linear-20              [-1, 1024]       4,195,328
           ReLU-21              [-1, 1024]               0
         Linear-22               [-1, 512]         524,800
           ReLU-23               [-1, 512]               0
         Linear-24                [-1, 10]           5,130
================================================================
Total params: 5,852,234
Trainable params: 5,852,234
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 3.49
Params size (MB): 22.32
Estimated Total Size (MB): 25.83
----------------------------------------------------------------
```

By using ReLU as activation function, 30 epochs and ADAM as optimizer

The testing data Loss with respect to number of epochs:



The testing data accuracy with respect to number of epochs:



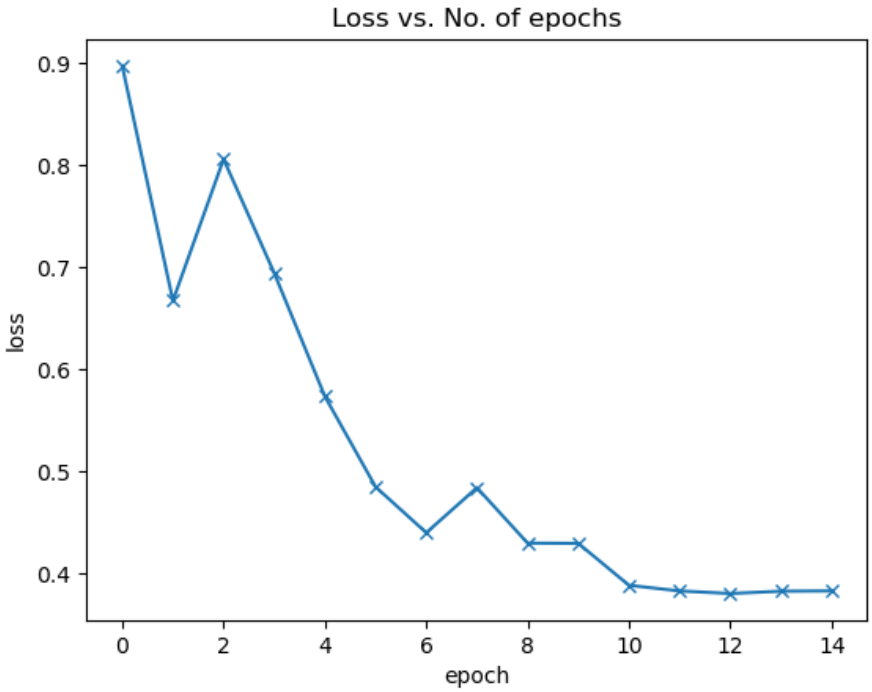
Here we can clearly observe that after 15'th epoch The loss started to increase. testing accuracy received for 15 epochs is: approximately 84.5 percent.
We can clearly observe the batch normalization increases classification accuracy.

Using RESNET architecture

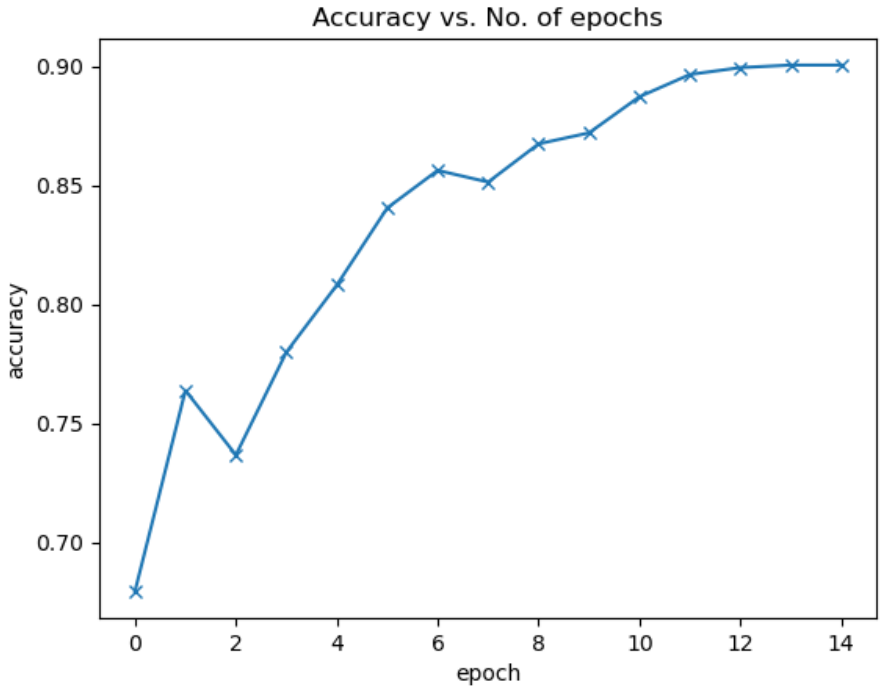
Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 32, 32]	1,792
BatchNorm2d-2	[-1, 64, 32, 32]	128
ReLU-3	[-1, 64, 32, 32]	0
Conv2d-4	[-1, 128, 32, 32]	73,856
BatchNorm2d-5	[-1, 128, 32, 32]	256
ReLU-6	[-1, 128, 32, 32]	0
MaxPool2d-7	[-1, 128, 16, 16]	0
Conv2d-8	[-1, 128, 16, 16]	147,584
BatchNorm2d-9	[-1, 128, 16, 16]	256
ReLU-10	[-1, 128, 16, 16]	0
Conv2d-11	[-1, 128, 16, 16]	147,584
BatchNorm2d-12	[-1, 128, 16, 16]	256
ReLU-13	[-1, 128, 16, 16]	0
Conv2d-14	[-1, 256, 16, 16]	295,168
BatchNorm2d-15	[-1, 256, 16, 16]	512
ReLU-16	[-1, 256, 16, 16]	0
MaxPool2d-17	[-1, 256, 8, 8]	0
Conv2d-18	[-1, 512, 8, 8]	1,180,160
BatchNorm2d-19	[-1, 512, 8, 8]	1,024
ReLU-20	[-1, 512, 8, 8]	0
MaxPool2d-21	[-1, 512, 4, 4]	0
Conv2d-22	[-1, 512, 4, 4]	2,359,808
BatchNorm2d-23	[-1, 512, 4, 4]	1,024
ReLU-24	[-1, 512, 4, 4]	0
Conv2d-25	[-1, 512, 4, 4]	2,359,808
BatchNorm2d-26	[-1, 512, 4, 4]	1,024
ReLU-27	[-1, 512, 4, 4]	0
MaxPool2d-28	[-1, 512, 1, 1]	0
Flatten-29	[-1, 512]	0
Linear-30	[-1, 10]	5,130

Total params: 6,575,370
Trainable params: 6,575,370
Non-trainable params: 0
Input size (MB): 0.01
Forward/backward pass size (MB): 9.07
Params size (MB): 25.08
Estimated Total Size (MB): 34.17

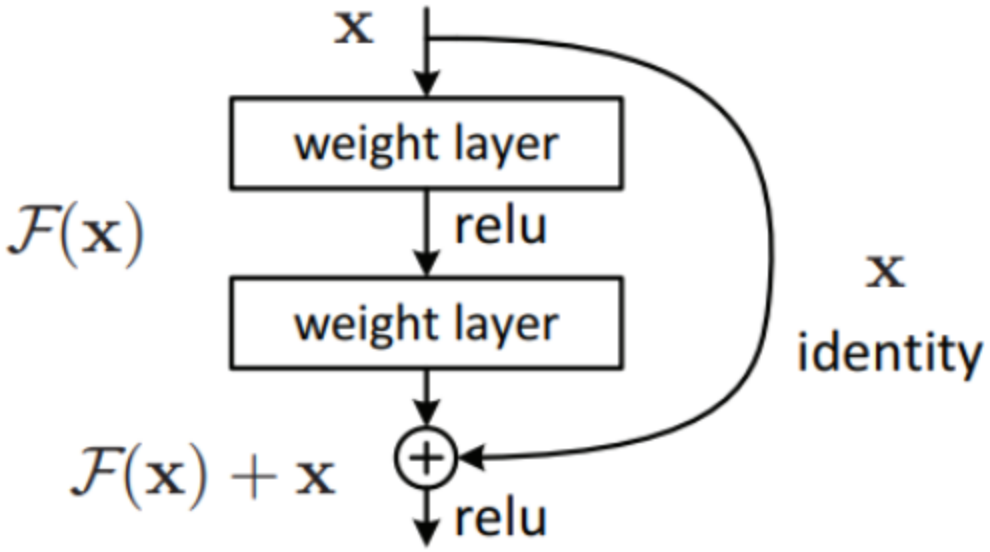
The testing data loss vs number of epochs:



The testing data accuracy vs number of epochs:



The testing data accuracy after 15 epochs is approximately 91 percent.



Skip (Shortcut) connection

The skip connection connects activations of a layer to further layers by skipping some layers in between. This forms a residual block. Resnets are made by stacking these residual blocks together. Clearly this is the best architecture for minimum loss entropy and maximum accuracy.
The time taken to run this is utmost 8-10 minutes.
When 15 epochs are ran it took approximately 8 minutes.

3b. **TRANSFER LEARNING**

The layers up until the last layers (fully connected ones) in CNNs such as AlexNet capture features and semantics of the images. If we train a CNN on large amounts of data for example ImageNet dataset. The convolutional layers have the information related to generic features observed across all the images, now we can use these features even to train a model on a different dataset. This is called transfer learning.

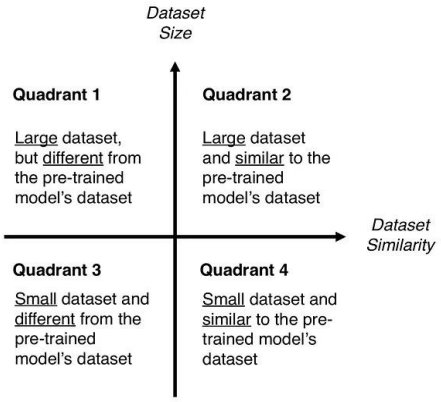
Transfer learning allows us to build accurate models in a timesaving way, this is because instead of starting the learning process from scratch we start from patterns that have been learnt while solving a different problem.

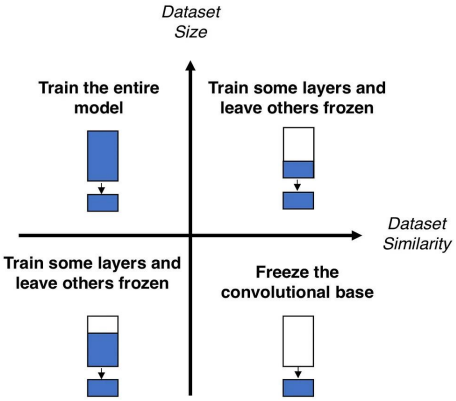
We have 3 different ways to repurpose our pre-trained model :

1. Train the entire model
2. Train some layers and leave the others frozen
3. Freeze the convolutional base

Dataset : The dataset we have chosen is the tensorflow flowers dataset - dataset.
Pre-trained CNN : The pre-trained CNN we have chosen to use for this exercise is VGG16. VGG16 is considered to be one of the best computer vision models to this date, the creators of this model evaluated the networks and increased the depth to using an architecture with very small filters. They pushed the depth to 16 layers making the number of parameters around 140 million. VGG16 is used for object detection and classification, it has been trained over 10 million+ images of 1000 different categories with a top-5 test accuracy of 92.7% and it is easy to use with transfer learning.

VGG16 has 13 Convolutional Layers, 5 Max pooling layers and 3 Dense layers, since it has a huge number of trainable parameters, the first way to repurpose the model i.e, to train the entire model again is of no use as it will take a very long time. We can either train some layers and leave the others frozen or we can freeze the convolutional base.





We can see that our problem comes under quadrant 4. So we remove the last layers (fully connected ones). Once we do this, we have essentially removed the classification layer that was trained on the ImageNet dataset. We also normalized the input data that is passed into the neural net.
The modified VGG16 model summary comes out to be :

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 150, 150, 3)]	0
block1_conv1 (Conv2D)	(None, 150, 150, 64)	1792
block1_conv2 (Conv2D)	(None, 150, 150, 64)	36928
block1_pool (MaxPooling2D)	(None, 75, 75, 64)	0
block2_conv1 (Conv2D)	(None, 75, 75, 128)	73856
block2_conv2 (Conv2D)	(None, 75, 75, 128)	147584
block2_pool (MaxPooling2D)	(None, 37, 37, 128)	0
block3_conv1 (Conv2D)	(None, 37, 37, 256)	295168
block3_conv2 (Conv2D)	(None, 37, 37, 256)	590080
block3_conv3 (Conv2D)	(None, 37, 37, 256)	590080
block3_pool (MaxPooling2D)	(None, 18, 18, 256)	0
block4_conv1 (Conv2D)	(None, 18, 18, 512)	1180160
block4_conv2 (Conv2D)	(None, 18, 18, 512)	2359808
block4_conv3 (Conv2D)	(None, 18, 18, 512)	2359808
block4_pool (MaxPooling2D)	(None, 9, 9, 512)	0
block5_conv1 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv2 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv3 (Conv2D)	(None, 9, 9, 512)	2359808
block5_pool (MaxPooling2D)	(None, 4, 4, 512)	0

Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688

Now, we take observations by using multiple models on top of the features extracted by the modified VGG16 net.

1. **Simple NN :**
   We use a custom neural network, which is in essence just the modified version of the fully connected layers that were already present in VGG16. After adding a flattening layer to the output of VGG16 net, we add three more layers -
   a. layer of 50 nodes and ReLU activation function.
   b. layer of 20 nodes and ReLU activation function.
   c. Prediction layer of 5 nodes and Softmax activation function.

We add these layers to the model and fit it on the training data. Upon evaluating the model on the test set we got an accuracy of 96%.

2. **Random Forests :**
   We choose a sample random forest with number of estimators (trees) = 50 and random state = 42. Now we extract the features by doing VGG16.predict(data) on train and test data respectively.

   These features are then reshaped(flattened) and the train features are the ones on which the Random Forest model is fit. Then we use the test features which were extracted and do RF_model.predict(test_features) to get the predicted test labels from the Random Forest model.

   Upon checking testing data accuracy we get a score of 100%

3. **SVM (Support Vector Machines) :**
   We take a sample SVM model, kernel = ‘rbf’ and gamma = ‘scale’. Similar to what we did with the Random Forest classifier, we fit the SVM classifier on the train features generated by the VGG16 net. The SVM classifier takes labels without the one hot encoding itself.

   Then we use the svm classifier to predict on the test features generated by the VGG16 net.

   Upon checking test data accuracy we get a score of 89%.
   Thus we can see that Random Forests classifier performs the best out of the tree for the specific case of this Dataset and using pretrained VGG16 as the neural net for transfer learning.

For Bike/Horse Classification :

We take the dataset as the bike/horse problem in assignment-2. Now we load the dataset and label the images simultaneously using a custom data loader function. We see that the bike/horse dataset comes under the category of quadrant 4 defined in the diagram shown earlier. So in this case we remove the last layers of VGGnet (fully connected ones). Once we do this, we have essentially removed the classification layer that was trained on the ImageNet dataset.

Now, we take observations by using multiple models on top of the features extracted by the modified VGG16 net.

4. **Simple NN :**
   We use a custom neural network, which is in essence just the modified version of the fully connected layers that were already present in VGG16. After adding a flattening layer to the output of VGG16 net, we add three more layers -
   d. layer of 50 nodes and ReLU activation function.
   e. layer of 20 nodes and ReLU activation function.
   f. Prediction layer of 2 nodes and Softmax activation function.

We add these layers to the model and fit it on the training data. Upon evaluating the model on the test set
We get - test loss, test acc: [0.17497128248214722, 0.9482288956642151]

5. **Random Forests :**
   We choose a sample random forest with the number of estimators (trees) = 25 and random state = 42. Now we extract the features by doing VGG16.predict(data) on train and test data respectively.

   These features are then reshaped(flattened) and the train features are the ones on which the Random Forest model is fit. Then we use the test features which were extracted and do RF_model.predict(test_features) to get the predicted test labels from the Random Forest model.

Upon checking testing data accuracy we get a score of 100%

6. **SVM (Support Vector Machines) :**
   We take a sample SVM model, kernel = ‘rbf’ and gamma = ‘scale’. Similar to what we did with the Random Forest classifier, we fit the SVM classifier on the train features generated by the VGG16 net. The SVM classifier takes labels without the one hot encoding itself.

   Then we use the svm classifier to predict on the test features generated by the VGG16 net.

Upon checking test data accuracy we get a score of 100%, the same as random forests.

3c. **YOLO V1 and YOLO V2**

YOLO V1:

- YOLOv1 (You Only Look Once version 1) is a real-time object detection system that was introduced in 2015. The system uses a single neural network to predict bounding boxes and class probabilities for objects in an image.
- The YOLOv1 network divides an input image into a grid of cells and each cell predicts a fixed number of bounding boxes and their corresponding object class probabilities. The network predicts the coordinates of the bounding boxes relative to the coordinates of the cell in which the box is located. The network also predicts class probabilities for each box and selects the box with the highest probability for each object.
- YOLOv1 is designed to be fast and efficient, allowing it to run in real-time on a GPU.The network uses a simple architecture consisting of 24 convolutional layers and 2 fully connected layers.
- In experiments, YOLOv1 achieved real-time detection speeds of up to 45 frames per second with a mean average precision (mAP) of 63.4% on the PASCAL VOC 2012 dataset. While it achieved lower accuracy than some other object detection systems, YOLOv1 was significantly faster and was able to process video streams in real-time.

YOLO V2:

- YOLOv2 (You Only Look Once version 2) is a real-time object detection system. It builds on the success of YOLOv1 by introducing several improvements to its architecture and training process.
- In experiments, YOLOv2 achieved state-of-the-art performance on several object detection benchmarks while maintaining real-time detection speed. It achieved a mean average precision (mAP) of 78.6% on the VOC 2012 dataset, significantly outperforming YOLOv1. YOLOv2 also introduced a smaller version of the network, called YOLOv2-tiny, which achieved real-time detection speeds of up to 244 frames per second with a mAP of 57.1% on the VOC 2007 dataset.
- YOLOv2 was introduced in 2016, which addressed some of the limitations of YOLOv1 and achieved higher accuracy.
- Here are **five additional features** of YOLOv2:
    1. Multi-scale training: YOLOv2 uses an approach called multi-scale training, which involves training the neural network on images of different sizes. This helps the network to detect objects of different sizes more accurately. YOLOv2 uses a technique called "passthrough" to combine the feature maps from different scales to improve detection accuracy.

2. Batch normalization: YOLOv2 uses batch normalization to improve the training process. Batch normalization normalizes the output of each layer to have zero mean and unit variance. This helps to stabilize the training process and speeds up convergence.
3. Anchor boxes: YOLOv2 introduces the concept of anchor boxes, which are pre-defined boxes of different sizes and aspect ratios that are used to detect objects. The network predicts the coordinates of the anchor boxes, rather than the coordinates of the objects directly. This helps to improve the accuracy of object detection, especially for objects of different sizes and shapes.
4. Darknet-19 architecture: YOLOv2 uses a new neural network architecture called Darknet-19, which has 19 convolutional layers. This architecture is deeper than the one used in YOLOv1, which had only 24 layers. The deeper architecture helps to improve the accuracy of object detection.
5. Intersection over Union (IoU) loss function: YOLOv2 uses a new loss function called the Intersection over Union (IoU) loss function, which penalizes the network based on the overlap between the predicted bounding boxes and the ground truth bounding boxes. This helps to improve the accuracy of object detection, especially for objects that are close together or partially occluded.

3d. **Object tracker: SORT + DeepSORT**

Object tracking is an important task in computer vision and has numerous applications, including surveillance, autonomous driving, and human-computer interaction. In this project, we will implement an object tracker using SORT and Deep SORT algorithms and apply it to a car counting task.

The tracker will be used to track objects in video frames and count the number of cars passing across a user defined line. To achieve this, we will use two popular object detection models, Faster RCNN and YOLO, to detect cars in the video frames. We will then use the detections made by these methods as input to the tracker to track the objects and count the number of cars that pass.

SORT (Simple Online and Realtime Tracking) is a popular algorithm for online tracking of objects in a video stream. It is a simple but effective algorithm that uses the Kalman filter to predict the next location of the object and the Hungarian algorithm to assign the detected objects to the predicted locations. This algorithm is fast and can work in real-time.

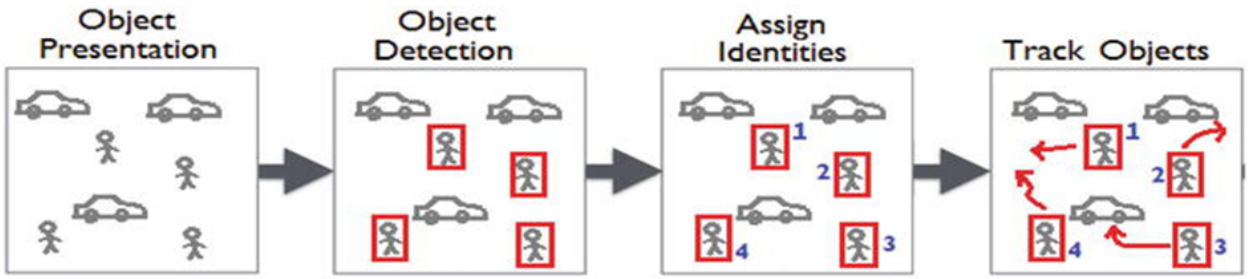
Deep SORT is an extension of SORT that uses a deep neural network to improve the tracking performance. It includes a feature extractor that extracts features from the object detections and a deep association network that learns to associate the detected objects with the tracked objects. This algorithm can handle occlusions and re-identification of the same object.

RCNN (Region-based Convolutional Neural Networks) is an object detection model that uses a combination of region proposal and convolutional neural networks. It first generates a set of region proposals using a selective search algorithm and then extracts features from each proposed region using a pre-trained CNN. These features are then fed into a set of fully connected layers that classify the object within the region and adjust the region proposal coordinates for better object localization. RCNN is an accurate object detection model, but it can be computationally expensive due to its multi-stage pipeline and can suffer from slow inference times. We use Faster RCNN to make the detections, in faster rcnn we overcome the disadvantages of using selective search and we also use another neural network pipeline that trains so that better regions of interest are chosen.

YOLO V2 is a real-time object detection model that predicts object classes and bounding box coordinates directly from the input image in a single pass. YOLO V2 is an improvement over the original YOLO model and uses a deep neural network with a series of convolutional layers followed by fully connected layers to make predictions. It divides the input image into a grid and for each grid cell, predicts the probability of each

class and the bounding box coordinates for any objects present in the cell. YOLO V2 is fast and efficient, making it well-suited for real-time object detection applications. However, it may not be as accurate as using faster RCNN in certain scenarios.

Basic Idea:



.

1. Building an object tracker using YOLO v2/ Faster RCNN and SORT:

a. Object Detection using YOLO v2:

YOLO v2 is a deep learning-based object detection algorithm that uses a single convolutional neural network to detect objects in real-time video streams. It divides an image into a grid of cells and predicts bounding boxes and class probabilities for each cell. The output of YOLO v2 is a list of bounding boxes that contain objects along with their class labels and confidence scores.

a. Object Detection using Faster RCNN:

Faster RCNN (Region-based Convolutional Neural Networks) is a deep learning-based object detection algorithm that uses a combination of region proposal networks and convolutional neural networks to detect objects in images and video streams. It generates a set of object proposals in each frame and classifies them using a CNN.

b. Object Tracking using SORT:

SORT (Simple Online and Real-time Tracking) is a simple and efficient algorithm for object tracking. SORT tracks objects by matching their bounding boxes across frames using a combination of distance metrics and Kalman filtering. It also handles occlusions, appearance changes, and track termination due to object disappearance.

c. Integrating detections and SORT:

To integrate the detections produced and SORT, we first use YOLO v2/Faster RCNN to detect objects in each frame of a video stream. We then feed the detected objects along

with their bounding boxes and confidence scores into the SORT algorithm. SORT uses these objects as initial detections and tracks them across frames using Kalman filtering and distance metrics

d. Updating Object Positions and IDs:

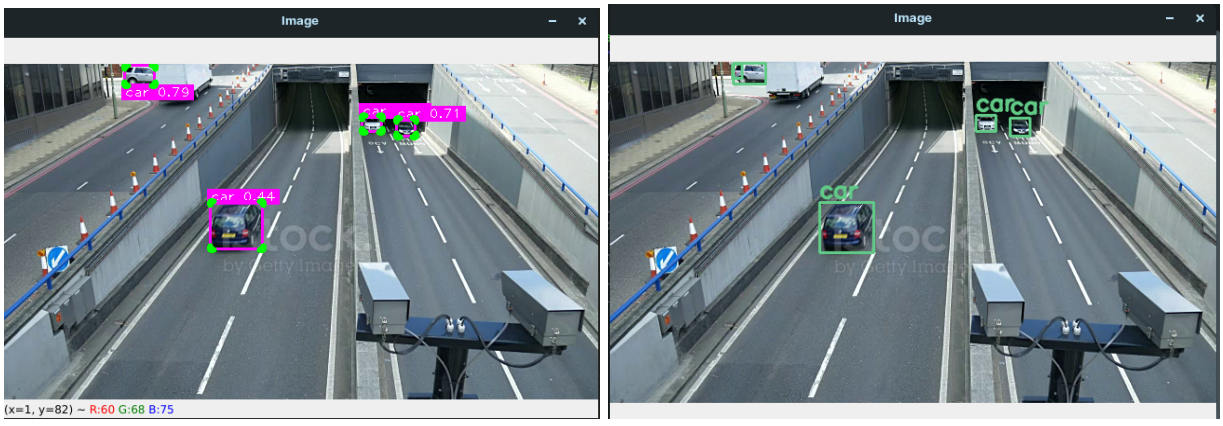
SORT updates the position and ID of each tracked object in each frame using the Kalman filter and distance metrics. It also handles new object detections and track terminations. Finally, it outputs a list of tracked objects along with their IDs and bounding boxes for each frame.

e. Drawing Bounding Boxes:

To visualize the tracked objects, we draw bounding boxes around them in each frame using their corresponding IDs and bounding boxes.

For **SORT** we have used “Alex Bewley’s” implementation forked from the github repository. The parameters that we have to define while creating an instance of the sort class are = max_age, min_hits, iou_threshold.

- Max-age : The number of frames for which an item is allowed to disappear. For example if a car of id=4 is not seen for 6 frames and the max-age defined is 5 then when the car is seen again it is given a new id.
- Iou-threshold : The intersection over union method threshold value that we accept. In order for the tracking boxes to be taken into consideration they have to pass the threshold.
- Min-hits : This is the minimum number of associated detections before which the track is initialized.



Above are the detection results with YOLO and Faster RCNN (left to right)

We define the tracker as “tracker = Sort(max_age=20,min_hits=3,iou_threshold=0.3)”.

model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True) for faster rcnn.

model = YOLO('../yolo-weights/yolov8n.pt') for yolo v8.

Once we pass the detections to the tracker, we get a set of ‘results’. The set contains multiple objects of type - [x1,y1,x2,y2,Id] where (x1,y1) and (x2,y2) are diagonally opposite coordinates and the Id is the Id assigned to the object which is being tracked.

We use these coordinates to draw the new bounding boxes, which come from the SORT tracker.

2. Building an object tracker using FasterRCNN/YOLO and Deep SORT:

a. Object Detection using Faster RCNN/YOLO:

Same as the previous step.

b. Object Tracking using Deep SORT:

Deep SORT is an extension of SORT that uses deep learning for feature extraction and similarity matching. It uses a deep appearance descriptor to track objects across frames, even when they are occluded or change appearance.

c. Integrating detections and Deep SORT:

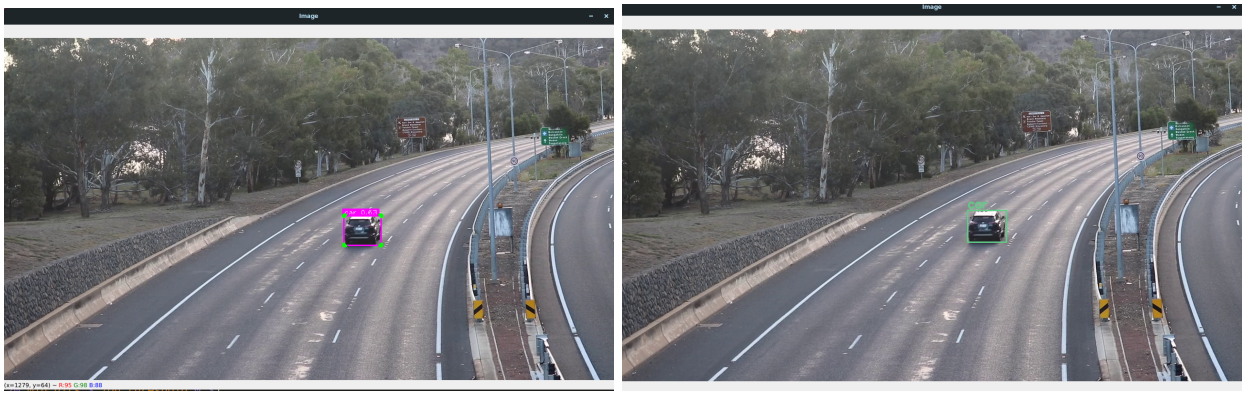
To integrate detections and Deep SORT, we first use either Faster RCNN or YOLO to detect objects in each frame of a video stream. We then feed the detected objects along with their bounding boxes and confidence scores into the Deep SORT algorithm. Deep SORT extracts deep appearance features from each detected object and uses them to track the object across frames using a combination of Kalman filtering and data association.

d. Updating Object Positions and IDs:

Deep SORT updates the position and ID of each tracked object in each frame using the Kalman filter and deep appearance features. It also handles new object detections and track terminations. Finally, it outputs a list of tracked objects along with their IDs and bounding boxes for each frame.

e. Drawing Bounding Boxes:

To visualize the tracked objects, we draw bounding boxes around them in each frame using their corresponding IDs and bounding boxes.



Here we have YOLO and Faster RCNN detections respectively on a new example video.

In summary, all four approaches involve detecting objects in each frame using a deep learning-based object detection algorithm (YOLO v2 or RCNN) and then tracking these objects across frames using a tracking algorithm (SORT or Deep SORT). The main difference is the level of sophistication of the tracking algorithm, with Deep SORT being the most advanced, as it uses deep learning for feature extraction and similarity matching.

Implementing a Car Counter :

Now that we have a tracking algorithm, we implement a simple car counter using the following steps :

1. We create a small circle at the center of each detected box. This circle will act as a representative point for the object (car).
2. We define a line that acts as the checkpoint. Whenever the center of the car crosses the checkpoint we increment the counter.

In order to make sure that we won't count the same car twice we increment the counter only when a new Id passes the line.

Note : Here we define a “band” essentially and we check for any frame if any center is present in the band. We do this so that we don’t miss out on any fast moving cars whose center is before the line in one frame and beyond the line in the succeeding frame.

With the usage of GPU, all methods are significantly faster, and the reason for using the yolo-tiny dataset instead of the large yolo is also the same.

Attached are the links of the demos -

Yolo-deepsort

yolo-sort

Note : On running faster rcnn with cpu the videos take a lot of time and therefore the recordings are too large.

SORT VS DEEP SORT :

Here are some of the main differences between SORT and DeepSORT:

1. Feature extraction: SORT uses hand-crafted features, such as color histograms and HOG features, to represent the appearance of objects in each frame. DeepSORT, on the other hand, uses a deep neural network to extract features from the objects, which are learned from a large dataset of labeled images. This allows DeepSORT to capture more complex and abstract visual features, making it more robust to changes in object appearance and occlusion.

2. Similarity matching: SORT uses a simple distance metric, such as Euclidean distance, to match objects across frames based on their appearance features. DeepSORT, in contrast, uses a learned similarity metric that takes into account both appearance and motion information to match objects. This allows DeepSORT to handle more challenging scenarios, such as partial occlusions, where appearance features alone may not be sufficient to distinguish between different objects.

3. Data association: SORT uses a simple data association method based on the Hungarian algorithm, which is computationally efficient but may produce suboptimal results in complex scenarios. DeepSORT, on the other hand, uses a more sophisticated data association method based on the Kalman filter, which can handle multiple object hypotheses and incorporate motion information to improve tracking accuracy.

4. ID management: SORT uses a simple ID management scheme that assigns a unique ID to each tracked object and assumes that the number of objects remains constant over time. DeepSORT, in contrast, uses a more robust ID management scheme that can handle object appearance changes and track terminations, and can re-assign IDs to objects that have been re-detected after a period of absence.

Overall, Deep SORT is a more advanced and sophisticated tracking algorithm than SORT, and can handle more challenging tracking scenarios with greater accuracy and robustness. However, it also requires more computational resources and may be slower than SORT in real-time applications.