

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
BANGALORE

VISUAL RECOGNITION
AI-825

Mini Project
Final Report

Karanjit Saha(IMT2020003)

Rishi Vakharia(IMT2020067)

Monjoy Narayan Choudhury(IMT2020502)

March 22, 2023

Task A

Preprocessing of Data

We initially applied data augmentation steps like horizontal flips and random cropping on images to increase the size of CIFAR 10 data-set and reduce the chance of over-fitting but it was increasing training time and not giving better results, so we removed that part.

NOTE:- We have tried 2 different architectures for this part.

Architecture I

Architecture of Network

Since it was expected that a medium deep learning network be used we decided to go with the following architecture:

1. (Input: 32x32x3 image) Conv1: With 64 3x3 kernels and padding = 1 (Output: 32x32x64)
2. Batch Normalisation
3. Activation
4. (Input: 32x32x64) MaxPool1: With 2x2 kernel and stride = 2 (Output: 16x16x64)
5. (Input: 16x16x64) Conv2: With 128 3x3 kernels and padding = 1 (Output: 16x16x128)
6. Batch Normalisation
7. Activation
8. (Input: 16x16x128) MaxPool2: With 2x2 kernel and stride = 2 (Output: 8x8x128)
9. Dropout1: With probability 0.05
10. (Input: 8x8x128) Conv3: With 256 3x3 kernels and padding = 1 (Output: 8x8x256)
11. Batch Normalization
12. Activation
13. (Output: 8x8x256) MaxPool3: With 2x2 kernel and stride = 2 (Output: 4x4x256)
14. (Input: 4x4x256) Flatten (Output: 1x4096)
15. Dropout2: With probability 0.1
16. (Input: 1x4096) Linear1: With 1024 neurons (Output: 1x1024)
17. Activation
18. (Output: 1x1024) Linear2: With 512 neurons (Output: 1x512)
19. Activation
20. Dropout3: With probability 0.1
21. (Output: 1x512) Linear3: With 10 neurons (Output: 1x10)

The result of this is the energies of the various class. For prediction taking the argmax of the resulting array should give us the class label. We could have gone with more linear layers and convolutional layers but we wanted to prevent recreating AlexNet or any *deep neural* network architecture as that would increase the training time and we had to run on multiple permutations of activation functions and optimizers which would not be possible to complete in on Kaggle or Colab.

Some Fixed Parameters

For the comparative study of different activation functions, we must naturally keep some hyper-parameters fixed to have some fair ground for comparison. This is the reason we decide to go ahead with the following set of hyper-parameters same for all the following observations

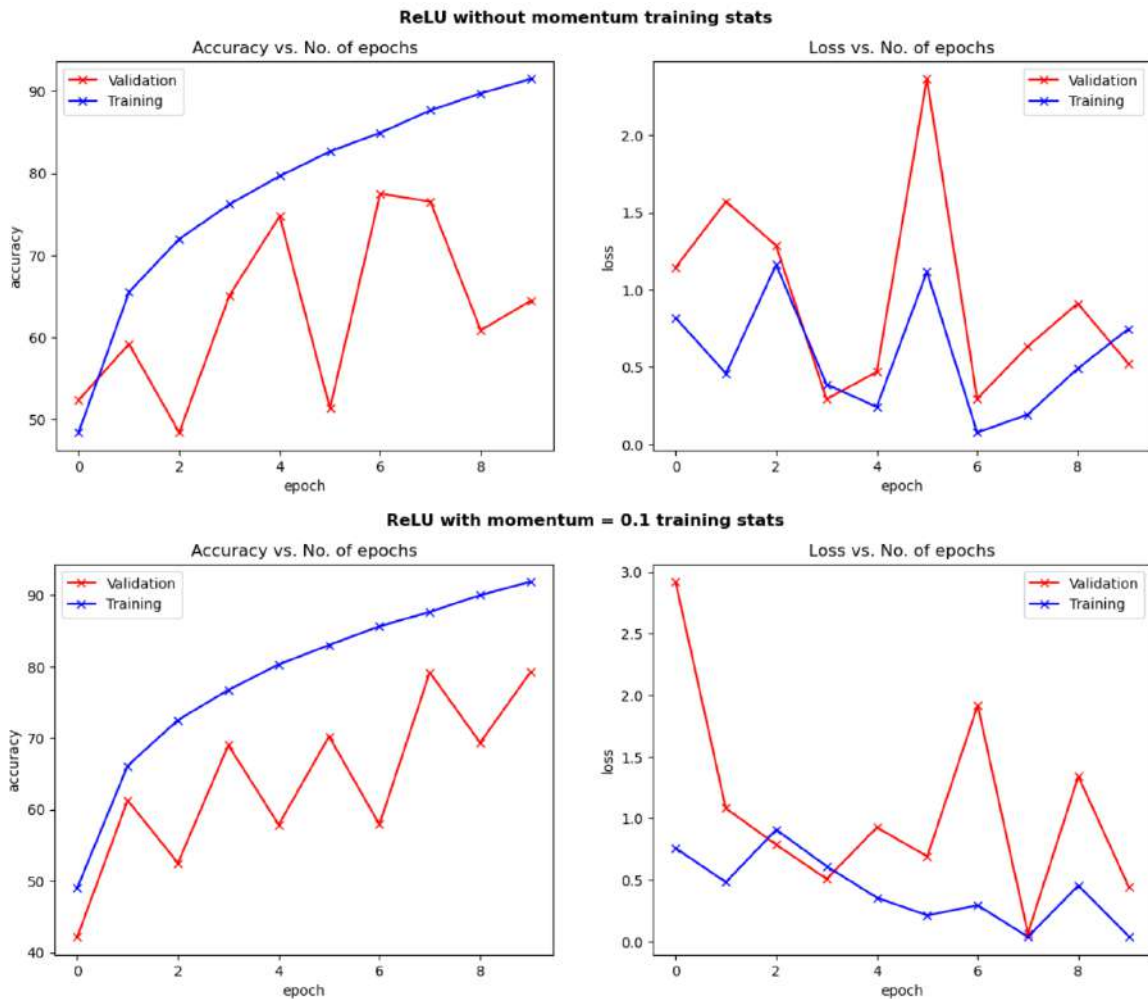
1. Learning Rate of Optimiser = 0.1 (We tried various other learning rates like 0.001, 0.01, 0.05 and 0.5, before selecting 0.1 as the learning rate.)
2. Batch Size = 64
3. Number of Epochs = 10

Results Obtained

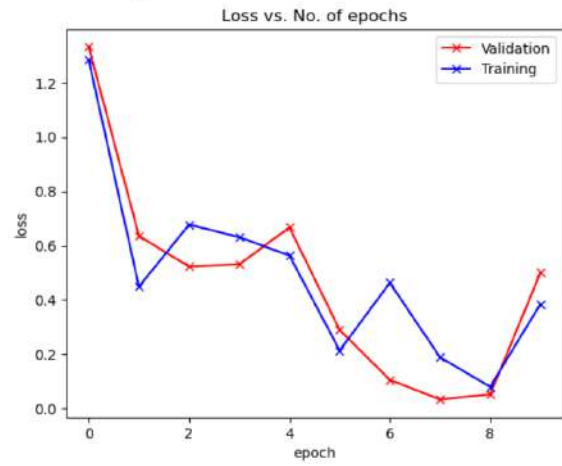
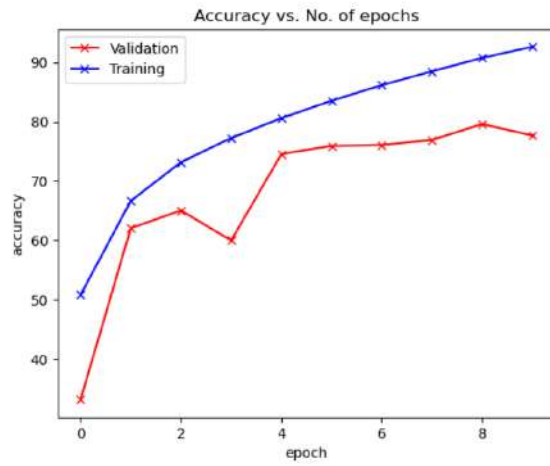
These are the results obtained on varying parameters like activation function, momentum & adaptive learning rate.

Note: Since we had to run a lot of permutations which would have taken a significant amount of time and would have exhausted our colab credits, we trained all the models on Kaggle by scheduling the notebook in their GPU. These are the values reported from there. We want to highlight the relative amount of time taken to train the model in the change of optimizers and activation function.

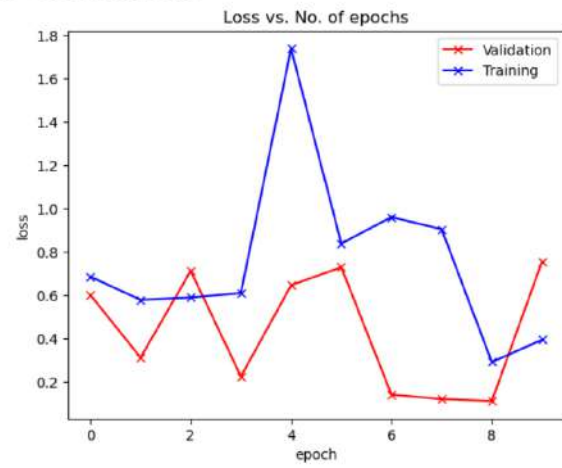
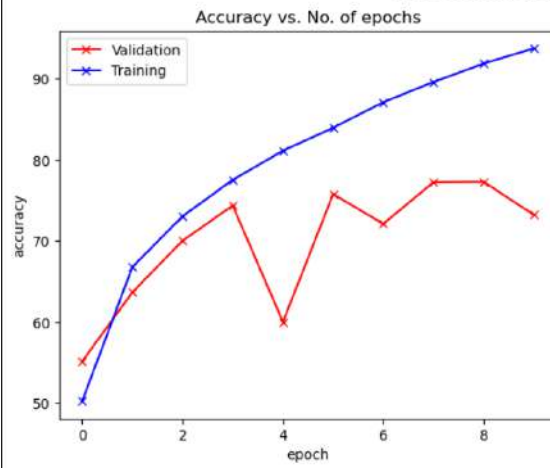
1. With ReLU activation function



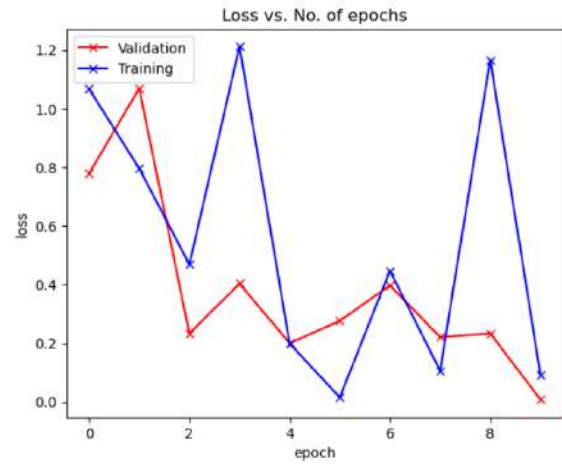
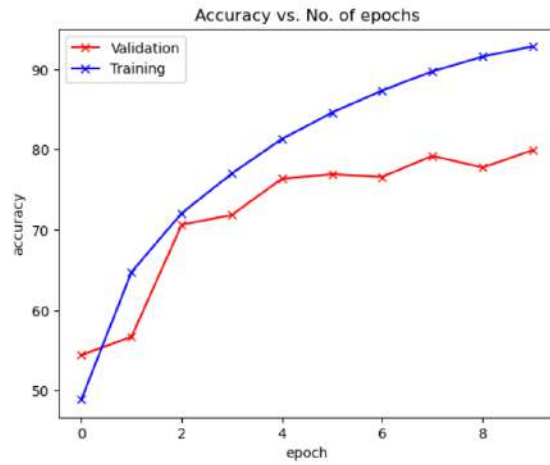
ReLU with momentum = 0.3 training stats

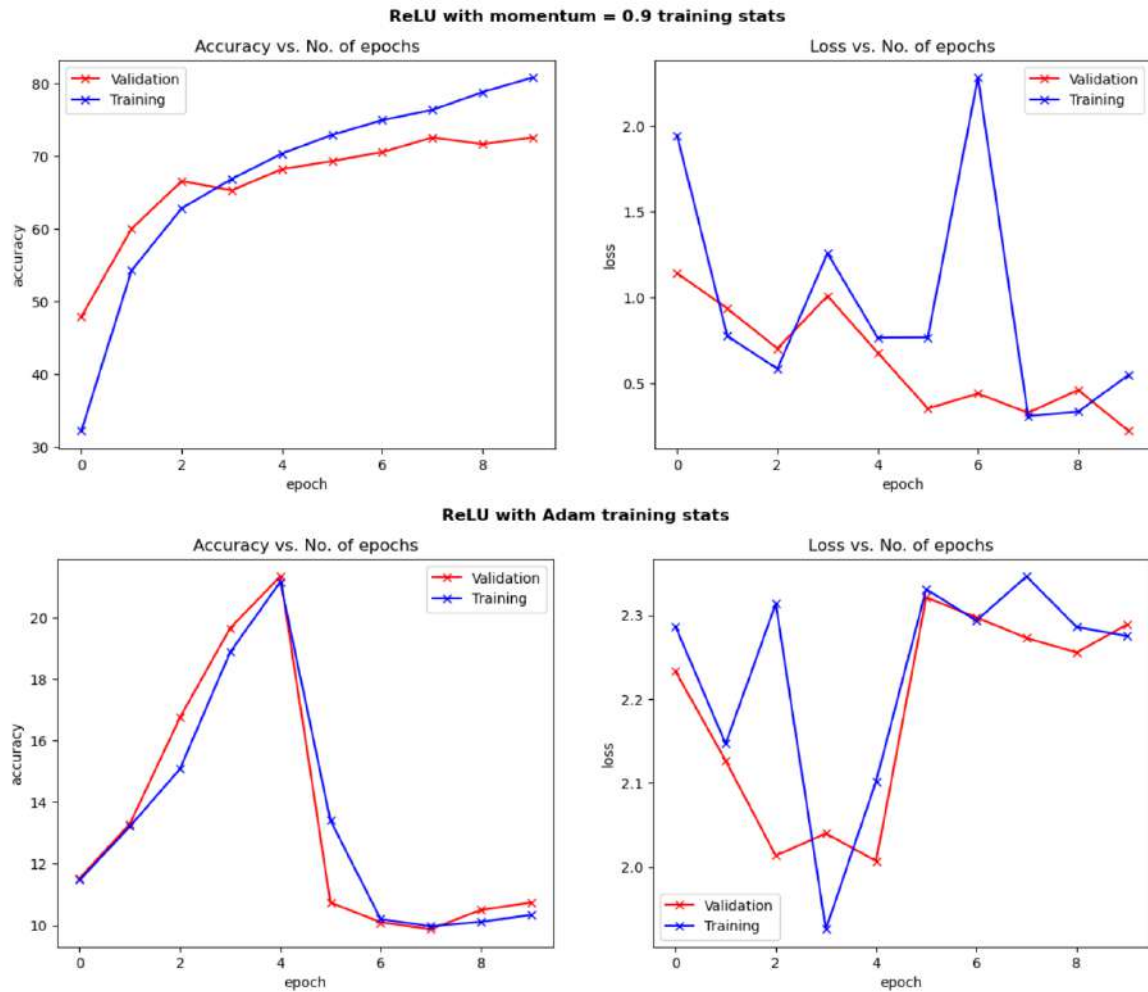


ReLU with momentum = 0.5 training stats



ReLU with momentum = 0.7 training stats





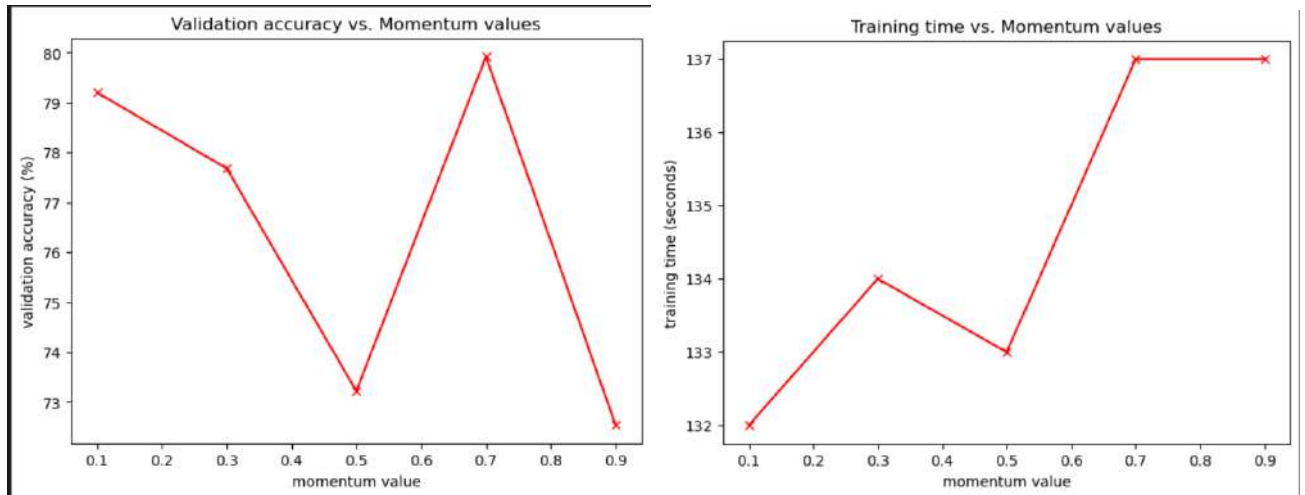
```
ReLU without momentum:
Training time: 0:02:21.143928
Validation accuracy: 64.48000073432922

ReLU with momentum 0.7:
Training time: 0:02:17.519034
Validation accuracy: 79.91999983787537

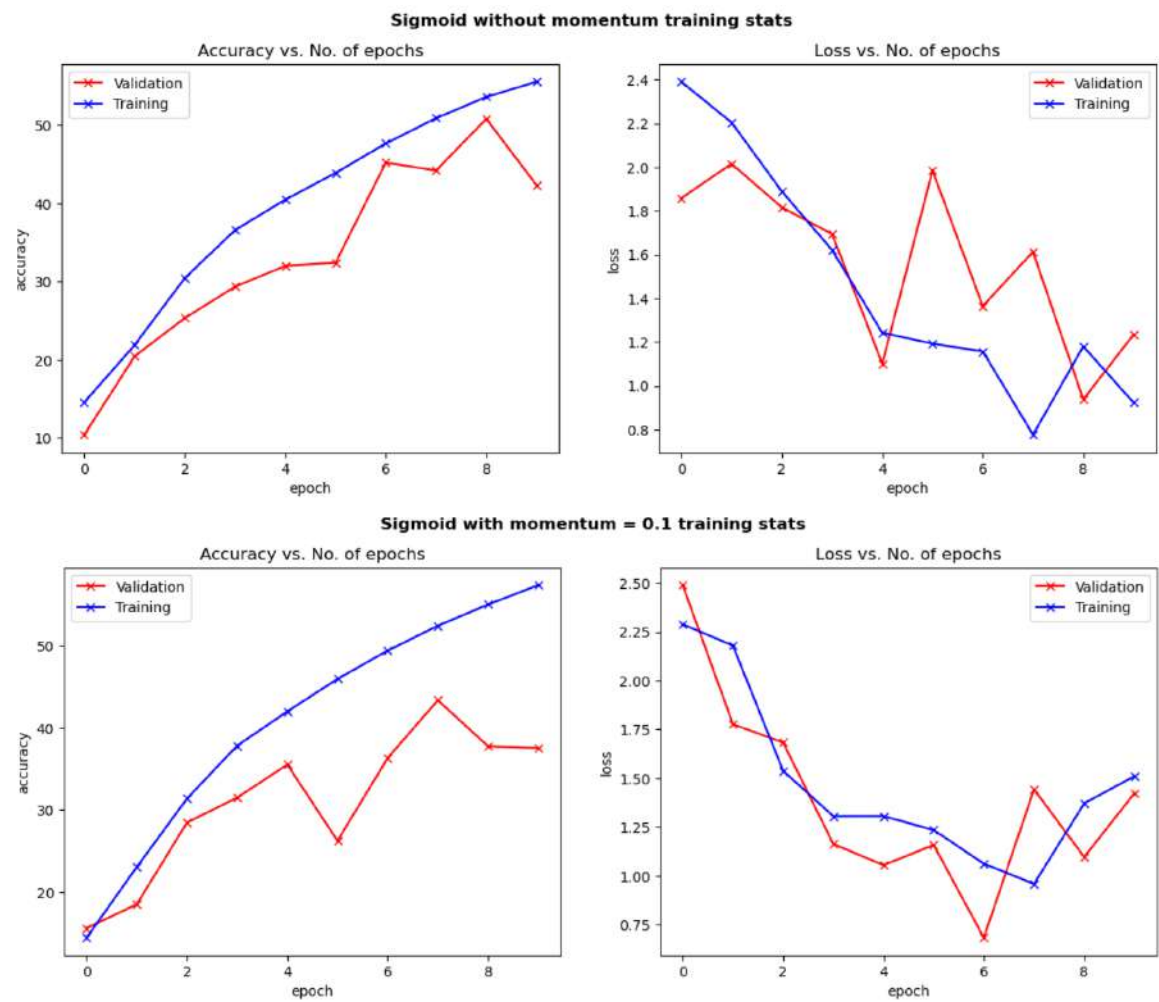
ReLU with Adam:
Training time: 0:02:26.021074
Validation accuracy: 10.740000009536743
```

How did we decide to take momentum = 0.7 ?

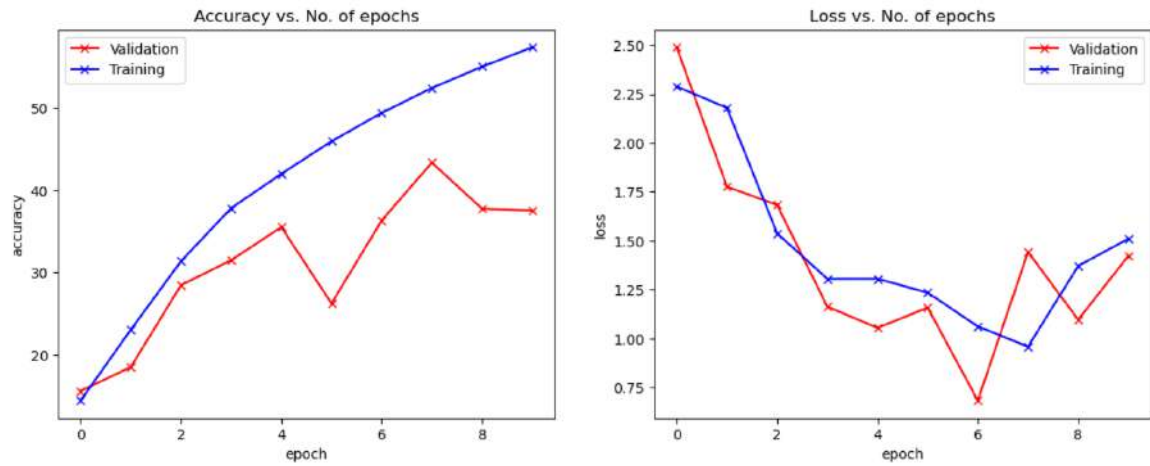
We tried different momentum values like [0.1, 0.3, 0.5, 0.7, 0.9] and 0.7 was giving the best results (as you can see in figures given below).



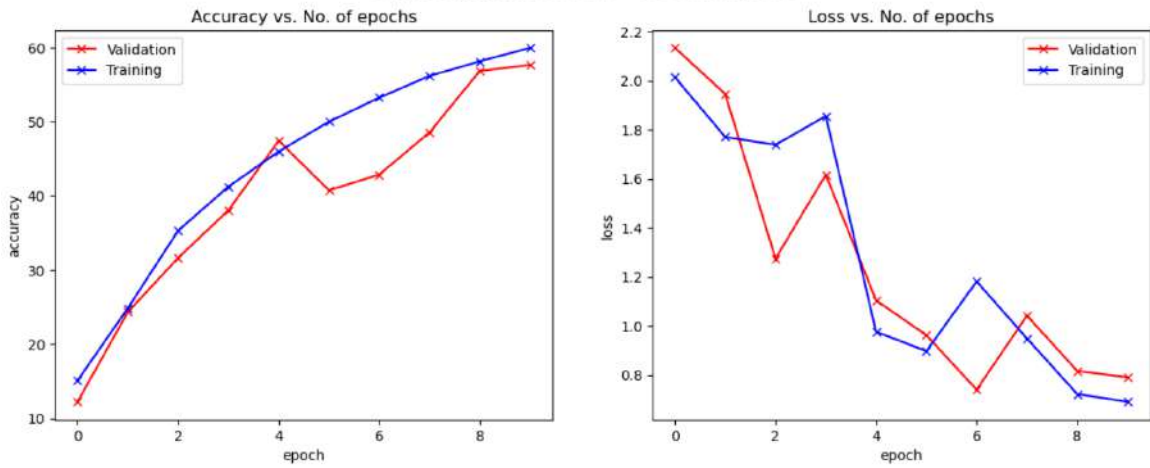
2. With Sigmoid activation function



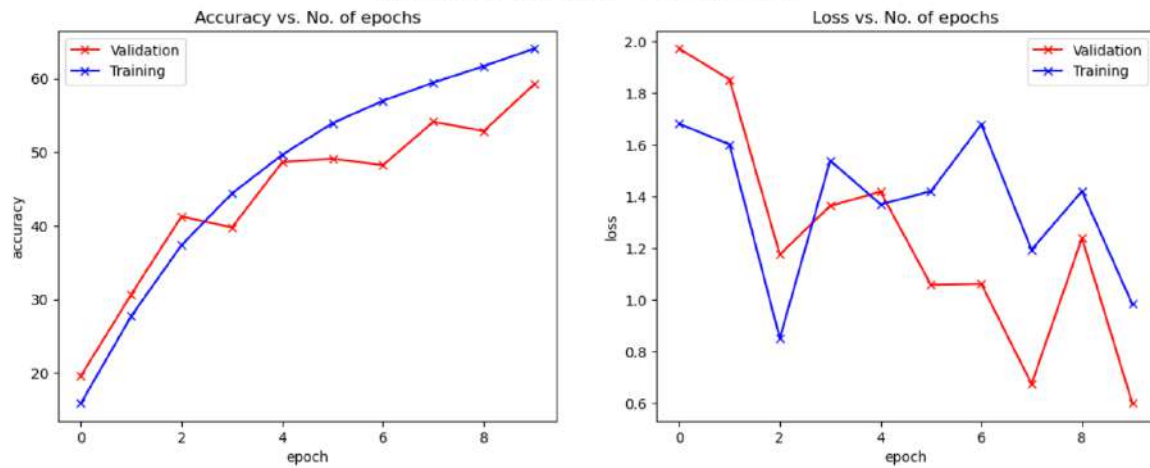
Sigmoid with momentum = 0.1 training stats



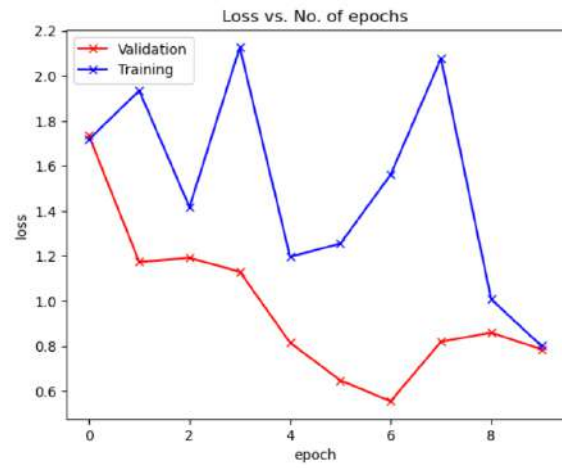
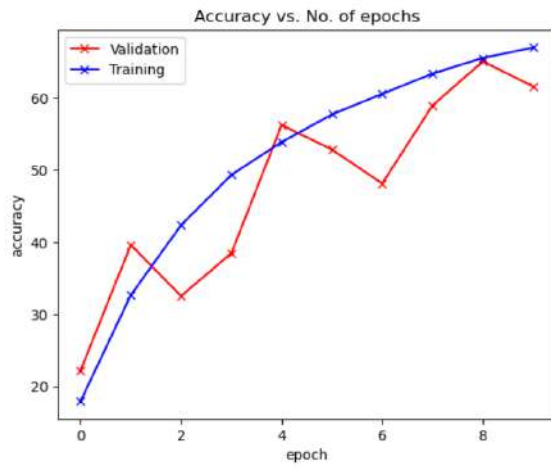
Sigmoid with momentum = 0.3 training stats



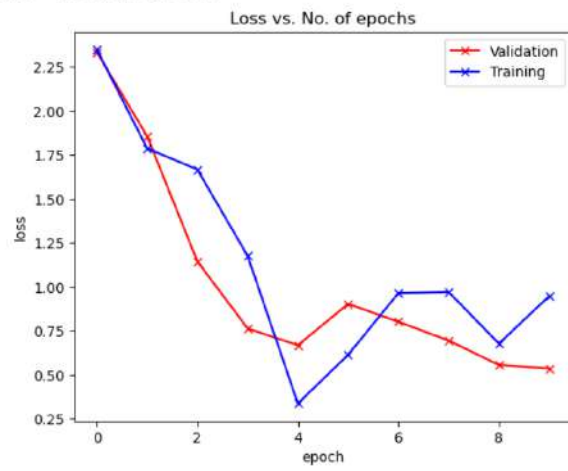
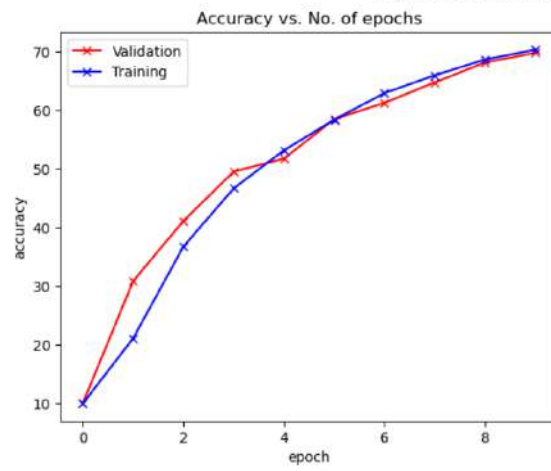
Sigmoid with momentum = 0.5 training stats



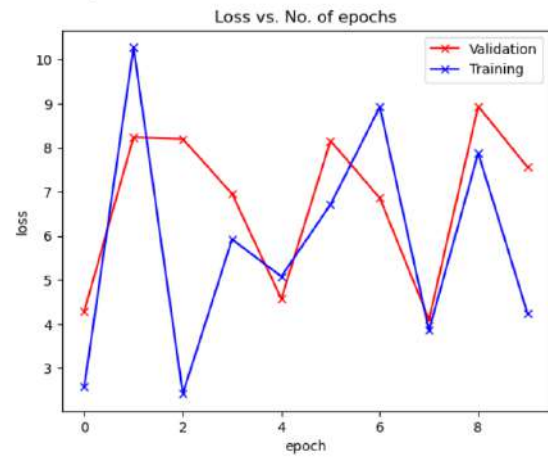
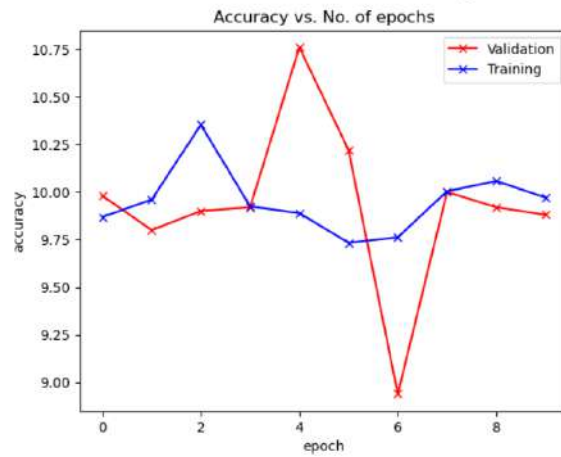
Sigmoid with momentum = 0.7 training stats



Sigmoid with momentum = 0.9 training stats



Sigmoid with Adam training stats




```

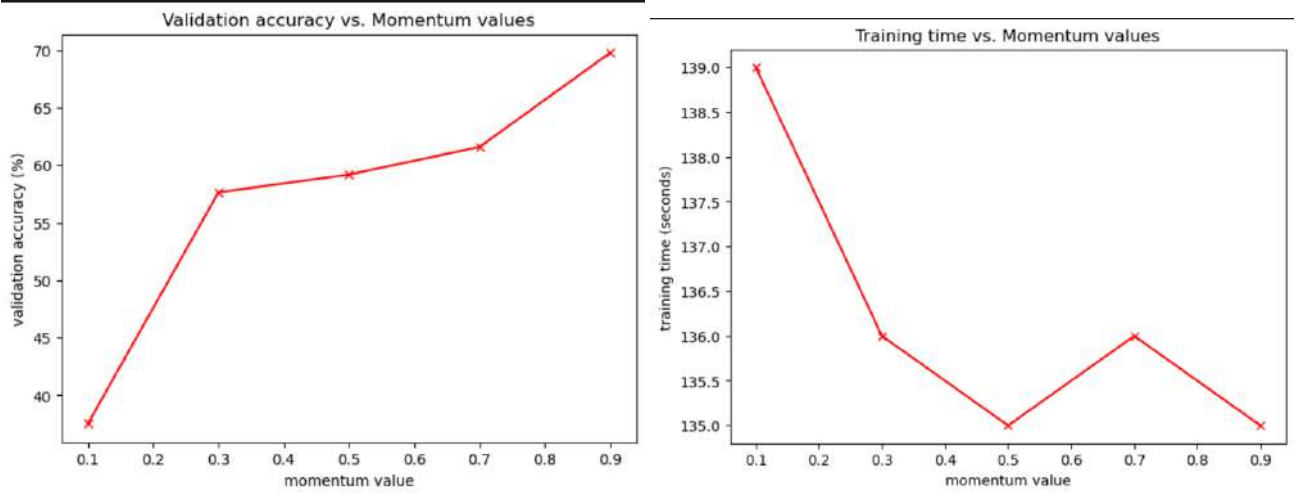
Sigmoid without momentum:
Training time: 0:02:17.945002
Validation accuracy: 42.319998145103455

Sigmoid with momentum 0.9:
Training time: 0:02:15.512908
Validation accuracy: 69.760000705719

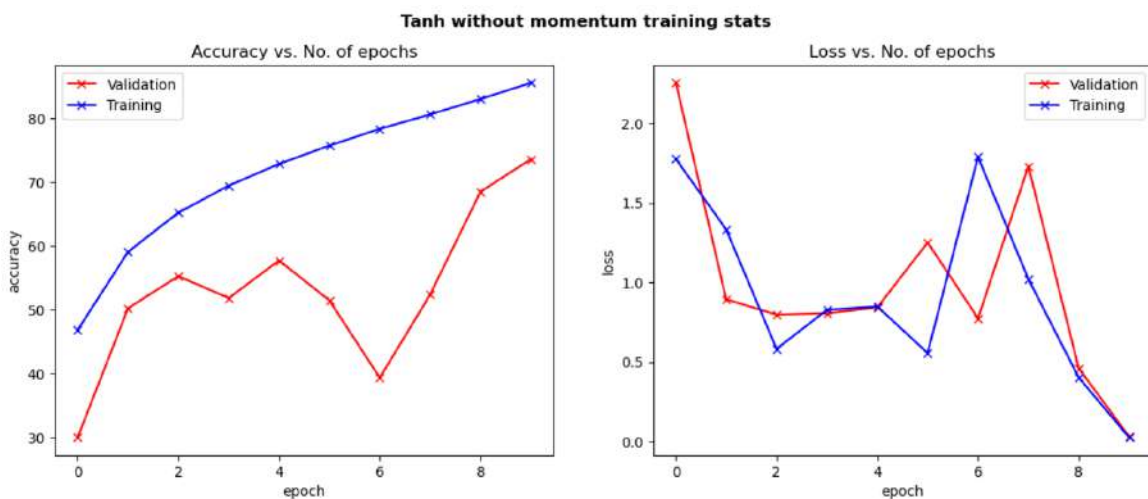
Sigmoid with Adam:
Training time: 0:02:21.437474
Validation accuracy: 9.879999607801437

```

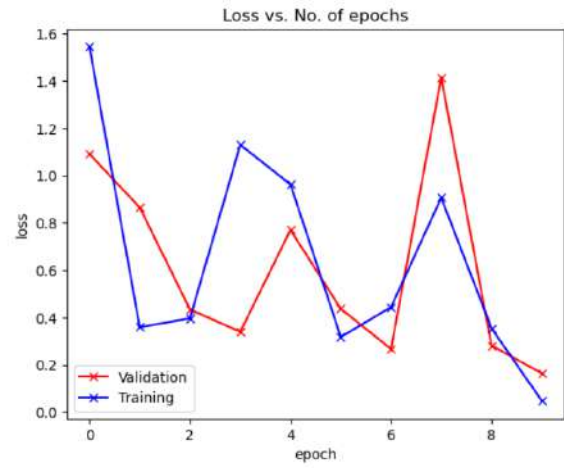
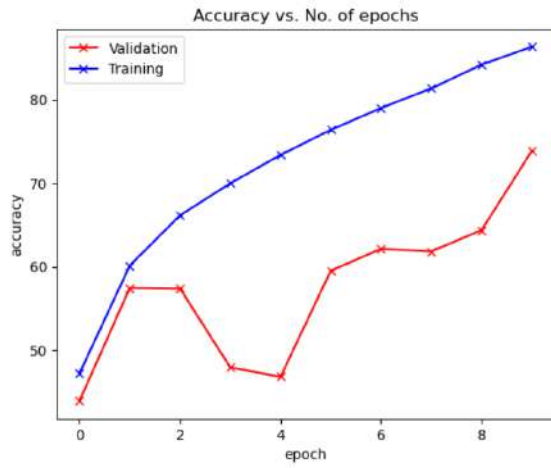
Here, momentum = 0.9 was giving the best results (figures given below).



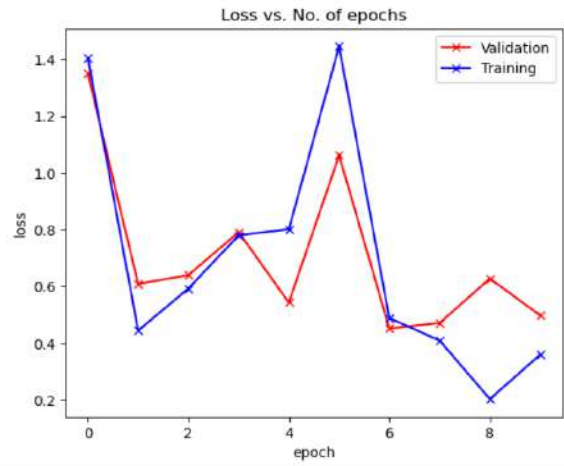
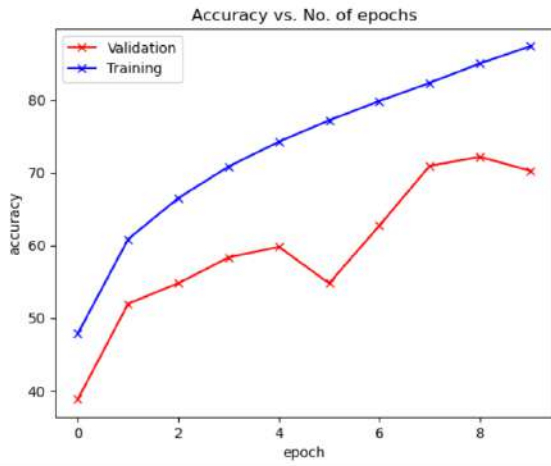
3. With Tanh activation function



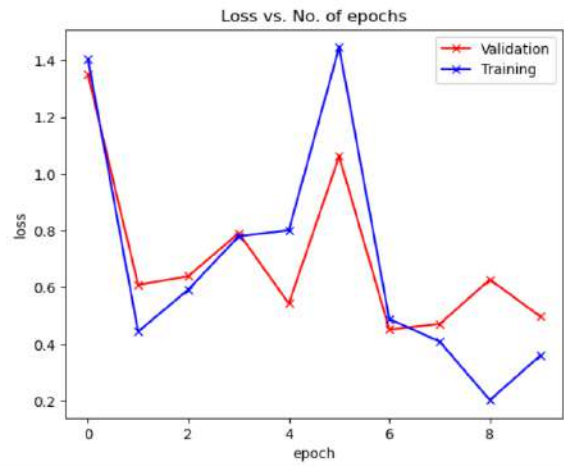
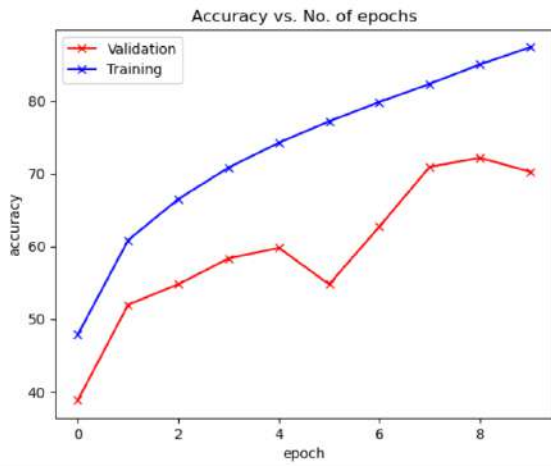
Tanh with momentum = 0.1 training stats



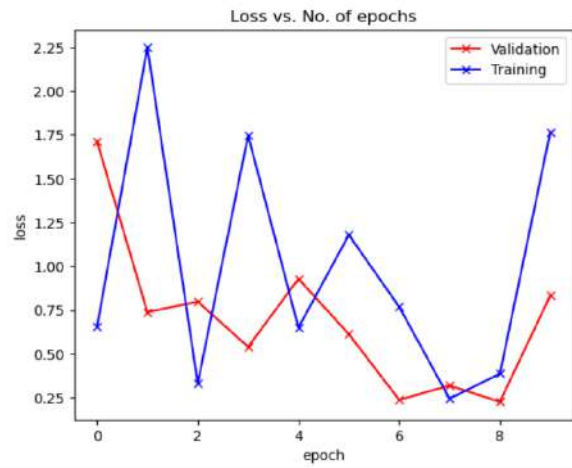
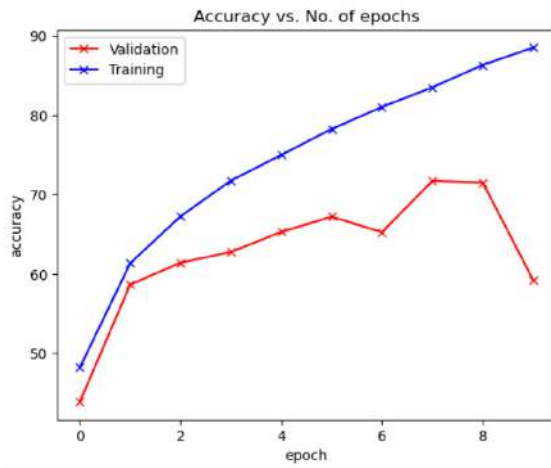
Tanh with momentum = 0.3 training stats



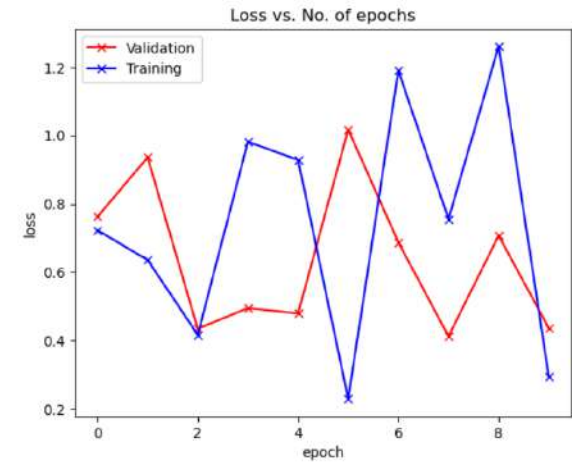
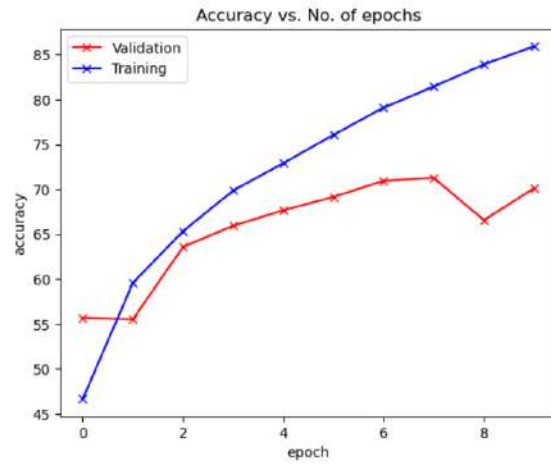
Tanh with momentum = 0.3 training stats



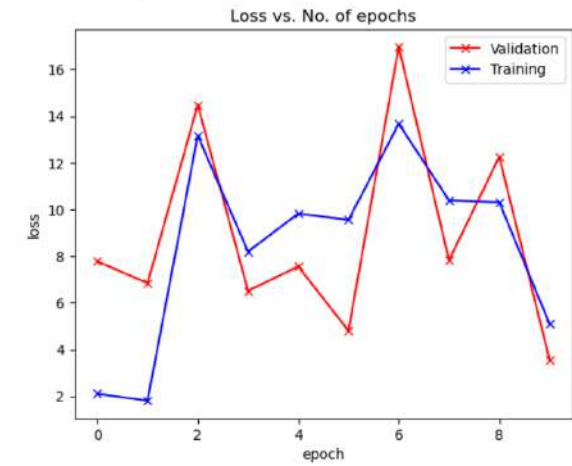
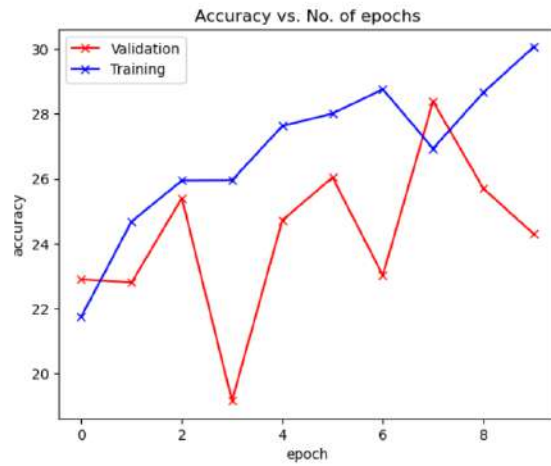
Tanh with momentum = 0.5 training stats

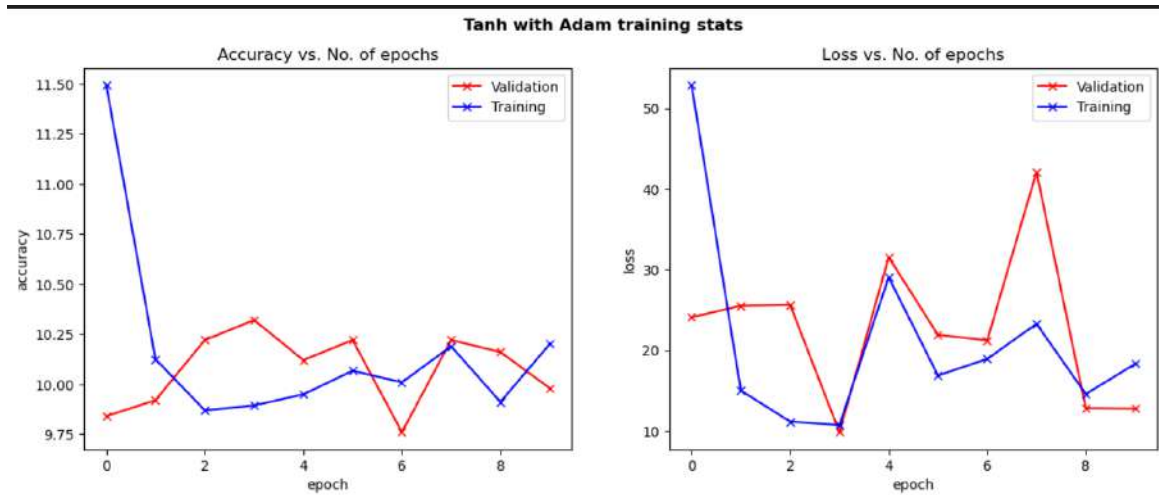


Tanh with momentum = 0.7 training stats



Tanh with momentum = 0.9 training stats





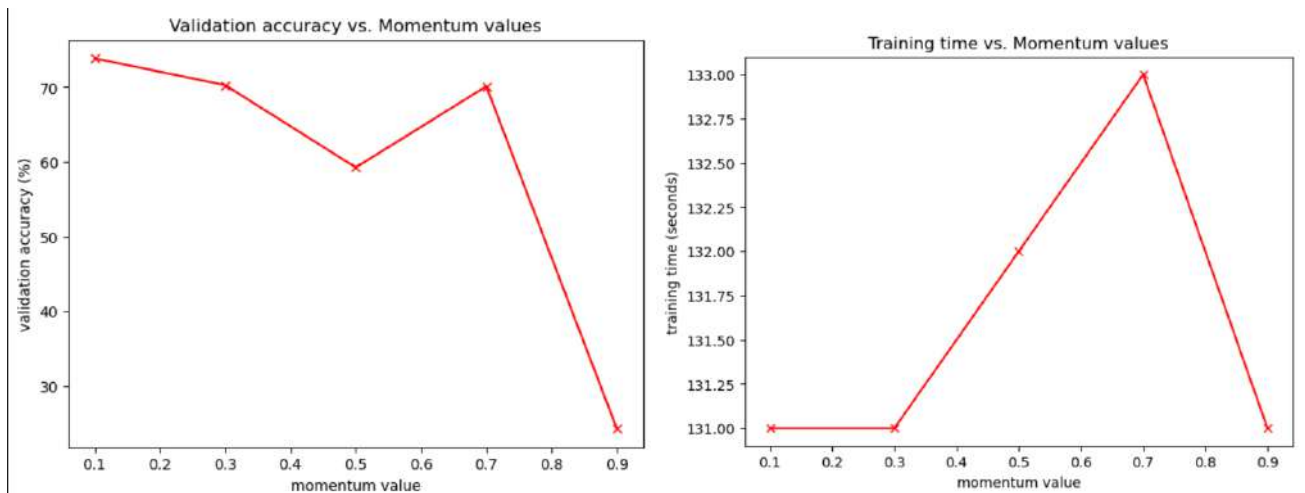
```

Tanh without momentum:
Training time: 0:02:10.552976
Validation accuracy: 73.51999878883362

Tanh with momentum 0.1:
Training time: 0:02:11.225336
Validation accuracy: 73.8599956035614

Tanh with Adam:
Training time: 0:02:21.535335
Validation accuracy: 9.97999981045723
  
```

Here, momentum = 0.1 was giving the best results (figures given below).



Observation

We find that on keeping

1. Activation function = ReLU
2. Momentum = 0.7
3. No adaptive gradient

we get the best results from this architecture which are:

1. Validation accuracy = 79.91
 2. Training time = 2 mins 17 seconds
- Now let's try the other architecture.

Architecture II

Architecture of Network

1. Conv1: A 3x3x32 filter with padding = 1
2. Maxpool1: with stride 2
3. Activation1: Used for comparison later
4. Conv2: 3x3x32 filter with padding = 1
5. Maxpool2: with stride 2
6. Linear1: which takes the feature size to 128.
7. Linear2: which takes the feature size to 10.

This gives energies of various classes. Taking argmax of it will give the predicted label. We defined a new architecture because we felt that the previous architecture was too complex and we wanted to experiment with a simpler architecture.

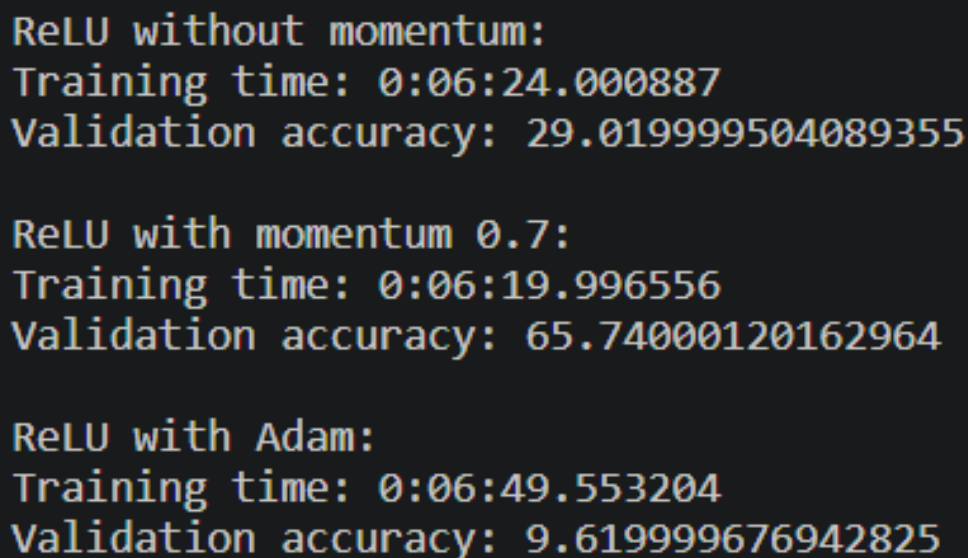
Some Fixed Parameters

1. Learning Rate of Optimiser = 0.1
2. Batch Size = 64
3. Number of Epochs = 10

Results Obtained

These are the results obtained on varying parameters like activation function, momentum & adaptive learning rate.

1. With ReLU activation function



```
ReLU without momentum:  
Training time: 0:06:24.000887  
Validation accuracy: 29.019999504089355  
  
ReLU with momentum 0.7:  
Training time: 0:06:19.996556  
Validation accuracy: 65.74000120162964  
  
ReLU with Adam:  
Training time: 0:06:49.553204  
Validation accuracy: 9.619999676942825
```

2. For Sigmoid activation function

```
Sigmoid without momentum:  
Training time: 0:06:56.847621  
Validation accuracy: 43.07999908924103  
  
Sigmoid with momentum 0.9:  
Training time: 0:06:57.742619  
Validation accuracy: 63.77999782562256  
  
Sigmoid with Adam:  
Training time: 0:07:46.740573  
Validation accuracy: 9.619999676942825
```

3. For Tanh activation function

```
Tanh without momentum:  
Training time: 0:07:15.387858  
Validation accuracy: 68.81999969482422  
  
Tanh with momentum 0.1:  
Training time: 0:07:21.511382  
Validation accuracy: 70.27999758720398  
  
Tanh with Adam:  
Training time: 0:06:59.032900  
Validation accuracy: 19.140000641345978
```

Observation

We find that this architecture is taking almost double training time as compared to architecture 1 even on keeping the parameters same. Also, the validation accuracies are not very different. Thus, we recommend on using Architecture 1. We didn't add the plots related to architecture 2 like the previous one to prevent the report from becoming unnecessarily long.

Recommended Architecture

Finally, our recommended architecture is Architecture 1 with

1. Activation function = ReLU
2. Momentum = 0.7
3. No adaptive gradient

which takes 2 mins and 17 seconds to train.

On testing this architecture on CIFAR-10's test dataset, we get an accuracy of **78.72 %**.

Task B

Dataset Used

For task B we have used the "Caltech 101" dataset. Caltech-101 consists of pictures of objects belonging to 101 classes, plus one background clutter class collected in September 2003 by Fei-Fei Li, Marco Andreetto, and Marc'Aurelio Ranzato. Each image is labeled with a single object. Each class contains roughly 40 to 800 images, totaling around 9000 images. Images are of variable sizes, with typical edge lengths of 200-300 pixels.

Preprocessing

We have split the dataset into train, test, and validation datasets. This is done to ensure that overfitting can be detected in the training stage itself. We further performed various preprocessing steps on this image like:-

1. Common Steps on both train and test:
 - (a) Resizing of Image to 224x224 to fit the AlexNet specifications.
 - (b) Normalising RGB channels with mean = [0.485, 0.456, 0.406] and standard deviation = [0.229, 0.224, 0.225].
2. Train Dataset:
 - (a) Random Rotations of images by 30°angle
 - (b) Random Horizontal Flips.
 - (c) Random Vertical Flips.

The extra data augmentation steps on the train dataset are done to make increase the data count for the classes having a low amount of images.

Hyperparameters

We have used Adam optimizer as optimizer for the network. We have used the following parameters to train our architecture.

- Learning Rate of Optimizer - 0.01
- Batch Size - 16
- Number of Epochs - 10

For Caltech101 Dataset

Architecture of Network - Alexnet

1. AlexNet model as a feature extractor: Pytorch has pre-trained models which made our job of importing it easier.
2. Classifier

Classifiers Used	Accuracy
LogisticRegression(max_iter=100000)	0.92
SVC(gamma='auto',kernel='linear')	0.87
SVC(gamma='auto',kernel='rbf')	0.71
RandomForestClassifier(max_depth=10, random_state=42)	0.59
RandomForestClassifier(max_depth=20, random_state=42, n_estimators=500)	0.74
KNeighborsClassifier(n_neighbors=5)	0.75

Table 1: Upon applying PCA(n_components=500)

Classifiers Used	Accuracy
LogisticRegression(max_iter=100000)	0.92
SVC(gamma='auto',kernel='linear')	0.55
SVC(gamma='auto',kernel='rbf')	0.31
RandomForestClassifier(max_depth=10, random_state=42)	0.43
RandomForestClassifier(max_depth=20, random_state=42,n_estimators=500)	0.52
KNeighborsClassifier(n_neighbors=5)	0.65

Table 2: Upon applying PCA(n_components=3000)

Classifiers Used	Accuracy
LogisticRegression(max_iter=100000)	0.92
SVC(gamma='auto',kernel='linear')	0.89
SVC(gamma='auto',kernel='rbf')	0.83
RandomForestClassifier(max_depth=10, random_state=42)	0.51
RandomForestClassifier(max_depth=20, random_state=42,n_estimators=500)	0.68
KNeighborsClassifier(n_neighbors=5)	0.63

Table 3: Upon using all the 9216 feautres

Architecture of Network - VGG net

1. VGG net model as a feature extractor: Pytorch has pre-trained models which made our job of importing it easier.
2. Classifier

Classifiers Used	Accuracy
LogisticRegression(max_iter=100000)	0.94
SVC(gamma='auto',kernel='linear')	0.90
SVC(gamma='auto',kernel='rbf')	0.75
RandomForestClassifier(max_depth=10, random_state=42)	0.66
RandomForestClassifier(max_depth=20, random_state=42,n_estimators=500)	0.84
KNeighborsClassifier(n_neighbors=5)	0.81

Table 4: Upon applying PCA(n_components=500)

Classifiers Used	Accuracy
LogisticRegression(max_iter=100000)	0.94
SVC(gamma='auto',kernel='linear')	0.60
SVC(gamma='auto',kernel='rbf')	0.33
RandomForestClassifier(max_depth=10, random_state=42)	0.48
RandomForestClassifier(max_depth=20, random_state=42,n_estimators=500)	0.62
KNeighborsClassifier(n_neighbors=5)	0.64

Table 5: Upon applying PCA(n_components=3000)

Classifiers Used	Accuracy
LogisticRegression(max_iter=100000)	0.94
SVC(gamma='auto',kernel='linear')	0.90
SVC(gamma='auto',kernel='rbf')	0.83
RandomForestClassifier(max_depth=10, random_state=42)	0.51
RandomForestClassifier(max_depth=20, random_state=42,n_estimators=500)	0.68
KNeighborsClassifier(n_neighbors=5)	0.63

Table 6: Upon using all the 25088 feautres

Therefore, the best architecture we found was:

1. VGG as a feature extractor.
2. Logistic Regression as a classifier.

For Bike vs Horse Dataset

Architecture of Network - Alexnet

1. AlexNet model as a feature extractor: Pytorch has pre-trained models which made our job of importing it easier.
2. Classifier

Classifiers Used	Accuracy
LogisticRegression(max_iter=100000)	1.0
SVC(gamma='auto',kernel='linear')	1.0
SVC(gamma='auto',kernel='rbf')	1.0
RandomForestClassifier(max_depth=10, random_state=42)	1.0
RandomForestClassifier(max_depth=20, random_state=42,n_estimators=500)	1.0
KNeighborsClassifier(n_neighbors=5)	1.0

Table 7: Upon using all the 9216 feautres

Architecture of Network - VGG net

1. VGG net model as a feature extractor: Pytorch has pre-trained models which made our job of importing it easier.
2. Classifier

Classifiers Used	Accuracy
LogisticRegression(max_iter=100000)	1.0
SVC(gamma='auto',kernel='linear')	0.94
SVC(gamma='auto',kernel='rbf')	0.74
RandomForestClassifier(max_depth=10, random_state=42)	1.0
RandomForestClassifier(max_depth=20, random_state=42,n_estimators=500)	1.0
KNeighborsClassifier(n_neighbors=5)	1.0

Table 8: Upon using all the 25088 feautres

NOTE:- We did not use PCA for bike vs horse classification since the data itself is very less, due to which the training time for the whole data itself is very less, unlike for the case of CalTech101 dataset.

Task C

The following are 5 additional features in YOLO v2 over YOLO v1:-

- **Change in Resolution of the Classifier**

YOLO v1 worked on 224x224 images. The newer version works on double the size of 448x448 images. Even though this change might look trivial, as per the figures, it increased the mean absolute precision by 4%.

- **Use of Darknet 19 as a backbone:**

YOLO v1 was based on 24 convolutional layers followed by 2 fully connected layers network. YOLO v2 improves upon this by using the Darknet 19 architecture which comprises 19 convolutional layers, 5 max-pooling layers, and a softmax layer for classification objects. Darknet is a neural network framework written in C language and CUDA and is faster than the previous version used.

- **Batch Normalization**

Batch normalization is a technique for training very deep neural networks that normalizes the contributions to a layer for every mini-batch. This has the impact of settling the learning process and drastically decreasing the number of training epochs required to train deep neural networks.

YOLO v2 normalizes the input layer by altering and scaling activations appropriately. This batch normalization results in the improvement of stability of the network as a result of which the MAP (mean average precision) has been improved by 2%. It also helps the model to regularise better and prevents overfitting.

- **Introduction of Anchor boxes**

In YOLO v1 the image is usually divided into grid cells and for each cell, we ran the inference for each cell. Two bounding boxes were predicted by each grid cell regardless of the scale of the image. This leads to an inflexibility in handling scale and aspect ratio and this is the issue that anchor boxes in YOLO v2 address. Anchor boxes are predefined bounding boxes that have a fixed size and aspect ratio placed in different parts of the image. During the training phase, the algorithm focuses on learning the amount to be changed (offset) to adjust the box according to the ground truth. Grid cells still exist and they assist in calculating the offset to the nearest anchor box and predicting the class.

- **Concept of Multi-Scale Training**

YOLO v1 upscaled each image from 224x224 to 448x448 and used a fixed input size. This led YOLO v1 to perform poorly when the objects were small or far from the camera. YOLO v2 tries to fix this by training images of different resolutions with base image quality resolution as mentioned above. This helped to introduce some sort of scale invariance of objects and increased accuracy in detecting objects in different scales and aspect ratio. This increased the practicality of using YOLO for real-world object detection.

Task D

Data Collection

We collected 3 videos from around Electronic City near traffic signals so that we can capture most of the cars coming for building our real-time car detector and counter.

Design Issues and Choices

1. Initially, we were instructed to use YOLO v2 as our base detection model and then follow it up with SORT/DeepSORT as the tracker. However when we tried to look for PyTorch-based compatible pre-trained weights and implementation and didn't find any. Upon communicating the same with the instructor, we were instructed to go ahead with any version of YOLO. Here we use ultralytics YOLO v5 model from the PyTorch hub for the YOLO part.
2. For the faster RCNN part, we got a pre-trained model from torch-vision itself.
3. We got a pre-implemented version of SORT from Github (links attached in the reference section).
4. Similarly we got a deepsort implementation from pip/ Github (link attached at the end).

Our Approach

We use the following steps, in general, to get the problem statement of car tracking done.

1. Break the video frame by frame
2. For each frame:
 - (a) Pass it through a detector (YOLOv5/Faster RCNN).
 - (b) Filter the detection which represents a car.
 - (c) On the filtered detection, pass them through an MOT tracker instance (SORT, DeepSORT)
 - (d) Increment the counter based on how sort and deepsort store the id's. In the case of sort, keeping track of the number of unique entries in an ID array would return the total number of cars given an ID. While in case of deepsort there is a readymade parameter that returns the largest id used so far which is technically based on the logic used by SORT.
 - (e) Note: We had tried typical data structure approaches where we stored the ids in a set and then returned a count but the results were the same as above.
 - (f) Annotate the details on the frame as required.
 - (g) Write frame to file.

Faster RCNN vs YOLO v5

Speedwise clearly YOLO v5 gives a better real-time frame-per-second output which allows us to apply it to video streams like cameras and create a product of value out of this project. However, we noticed that Faster RCNN gives better overall detection of on-screen objects especially cars which are far away in the frame.

SORT vs DeepSORT

Both algorithms try to achieve tracking of objects. Some of the differences we noticed in the two are as follows:

1. SORT uses the concept of velocity and filters to detect it i.e uses the appearance and motion while Deep SORT uses a CNN as a feature extractor along with Kalman filters.
2. Realtime wise for online videos i.e if we are rendering on the fly, SORT gives a better result (which is substantial on devices with sub-standard GPU) as it doesn't have the overhead of going through a forward pass of a neural network. In our case this overhead adds up due to the 2 forward passes of the neural network: once for the detector (Faster RCNN/YOLO) and the other in the case of DeepSORT.

3. However the performance aspect in case of occlusion can be better tackled using a CNN which leads to the DeepSORT being better to track in some cases. Also when the screen is too crowded, DeepSORT tracks each element better.
4. Considering the tradeoff between real-time and accuracy we could come up with the following conclusion:
 - (a) For real-time/ online detection on commodity hardware, we feel that using SORT would be a better option as anyways accuracy usually is affected due to the online nature of the data (frame loss and compression).
 - (b) For cases where accuracy is much more of a concern rather than the frame rates received, using the Deep learning-powered DeepSORT.

Problems faced in implementation

1. Lack of GPU credits: To get close to the acceptable performance we had to try different hyper-parameters of SORT. For each output, we required a substantial amount of GPU memory and free services got exhausted easily, hence we had to get the best result from whatever we had tried.
2. Generalising hyper-parameter to perform well enough for each test case: We noticed that for some sets of parameters, some videos gave a better output than the rest. We tried to find a single parameter for the 3 test cases we had which gave close to the right results.
3. Dependency handling: The original repositories for SORT and DeepSORT made as proof of work for the papers is ill-maintained with a lot of outdated packages. We had to manually resolve the issues in a separate environment.
4. Lack of documentation on SORT and DeepSORT often made deciding and understanding the hyper-parameters tough.

Hyper-parameters used

We try out 4 combinations viz: 1. Faster RCNN + SORT, 2. Faster RCNN + DeepSORT, 3. YOLO v5 + SORT and 4. YOLO v5 + DeepSORT. We use the following parameters in each of the cases

1. `confidence_threshold = 0.95` in Faster RCNN,
`object_tracker = Sort(max_age=2800, min_hits=3)`
2. `confidence_threshold = 0.95`,
`DeepSort(max_iou_distance=0.2,max_age=250,nms_max_overlap =0.05,gating_only_position=True, n_init=2,max_cosine_distance=0.9).`
 These parameters usually handle suppression and the number of frames to track an object even after that object is not in the scene and attempt to accommodate depth or not.
3. `confidence_threshold = 0.6`, `object_tracker = Sort(max_age=2800, min_hits=3)`
4. `confidence_threshold = 0.6`,
`object_tracker = Sort(max_age=2800, min_hits=3)`
5. `confidence_threshold = 0.95`,
`DeepSort(max_iou_distance=0.2,max_age=250,nms_max_overlap =0.05,gating_only_position=True n_init=2,max_cosine_distance=0.9).`

Observations

We didn't have any pre-labeled ground truth values as such. So we decided to manually count the cars which we see by eye. Therefore whatever value we report as the ground truth itself can consist of human errors. Keeping that in mind we get the following statistics for the combinations defined:

1. Video 1: Visually seen cars (by us) - 9
 - (a) Counter for Faster RCNN + SORT: 10

- (b) Counter for Faster RCNN + DeepSORT: 13
 - (c) Counter for YOLO v5 + SORT: 11
 - (d) Counter for YOLO v5 + DeepSORT: 12
2. Video 2: Visually seen cars (by us) - 14
- (a) Counter for Faster RCNN + SORT: 14
 - (b) Counter for Faster RCNN + DeepSORT: 14
 - (c) Counter for YOLO v5 + SORT: 15
 - (d) Counter for YOLO v5 + DeepSORT: 12
3. Video 3: Visually seen cars (by us) - 25
- (a) Counter for Faster RCNN + SORT: 26
 - (b) Counter for Faster RCNN + DeepSORT: 31
 - (c) Counter for YOLO v5 + SORT: 42
 - (d) Counter for YOLO v5 + DeepSORT: 38

Some Comments

1. We notice that the auto is being falsely detected as a car. This is due to the lack of context in the data used in training the detection model. The models are trained using datasets that definitely don't have data of autos as a separate category.
2. In the third video, the results are extremely poor. This might be due to blur and heavy occlusion. Also, cars come from various directions (left and right lanes as well as an intersection of two roads) along with the video source is unstable.
3. However, the results in the remaining two videos are good considering it is difficult for a human itself to label this data.
4. We can improve this further (for an end-to-end project) by using a line of reference to increment the counter so that the counter abruptly doesn't increase whenever it detects a car. This can be used as a traffic regulation solution itself.

Link to output videos

The input and output videos can be found [here](#).

References

1. Caltech 101 Dataset:
<https://data.caltech.edu/records/mzrjq-6wc02>
2. Caltech 101 and Transfer Learning:
<https://debuggercafe.com/getting-95-accuracy-on-the-caltech101-dataset-using-deep-learning/>
3. Building custom image transformations for datasets:
<https://stackoverflow.com/questions/62371522/pytorch-how-to-apply-another-transform-to-an-existing-dataset>
4. YOLO:
YOLO v1 - <https://arxiv.org/abs/1506.02640>
YOLO v2 - <https://arxiv.org/abs/1612.08242>
Comparative Study - <https://medium.com/@venkatakrishna.jonnalagadda/object-detection-yolo-v1-v2-v3-c3d5eca2312a>
5. Faster RCNN for object detection:
<https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e>
6. OpenCV + Faster RCNN:
<https://learnopencv.com/faster-r-cnn-object-detection-with-pytorch/>
7. Torchvision documentation for pre-trained models, sklearn and Pytorch documentation.
8. SORT implementation: <https://github.com/abewley/sort>
9. DeepSORT implementation: <https://pypi.org/project/deep-sort-realtime/>
10. YOLO v5 + DeepSort: <https://www.youtube.com/watch?v=IuVnYfg4vPQ>